aws

Developer Guide

Amazon DynamoDB

API Version 2012-08-10

Copyright © 2026 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

Amazon DynamoDB: Developer Guide

Copyright © 2026 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

Amazon's trademarks and trade dress may not be used in connection with any product or service that is not Amazon's, in any manner that is likely to cause confusion among customers, or in any manner that disparages or discredits Amazon. All other trademarks not owned by Amazon are the property of their respective owners, who may or may not be affiliated with, connected to, or sponsored by Amazon.

Table of Contents

What is Amazon DynamoDB? ... 1
Characteristics ... 2
Serverless ... 2
NoSQL ... 2
Fully managed ... 2
Single-digit millisecond performance at any scale ... 2
Use cases ... 3
Capabilities ... 4
Multi-active replication with global tables ... 4
ACID transactions ... 4
Change data capture ... 4
Secondary indexes ... 5
Service integrations ... 5
Serverless integrations ... 5
Importing and exporting data to Amazon S3 ... 5
Zero-ETL integration ... 6
Caching ... 6
Security ... 6
Resilience ... 7
Global tables ... 7
Continuous backups and point-in-time recovery ... 7
On-demand backup and restore ... 8
Accessing DynamoDB ... 8
Pricing ... 8
Getting started ... 8
Getting started with DynamoDB ... 10
First-time user resources ... 11
Amazon DynamoDB additional best practices for first-time users ... 12
AWS CLI resources ... 12
Programming resources ... 13
Accessing DynamoDB ... 13
Using the console ... 14
Using the AWS CLI ... 14
Using the API ... 17

Using the NoSQL workbench ... 17
IP address ranges ... 18
Dual-stack endpoints for Internet Protocol version 6 (IPv6) ... 18
Prerequisites ... 19
Setting up DynamoDB ... 19
Setting up DynamoDB (web service) ... 20
Setting up DynamoDB local (downloadable version) ... 24
Step 1: Create a table ... 48
Step 2: Write data ... 102
Step 3: Read data ... 134
Step 4: Update data ... 161
Step 5: Query data ... 193
Step 6: (Optional) clean up ... 232
Next steps ... 250
Console-to-Code ... 250
How it works ... 251
Benefits of using Console-to-Code with DynamoDB ... 251
Example use cases ... 252
Getting started ... 252
How it works ... 253
Cheat sheet ... 253
Initial setup ... 253
SDK or CLI ... 254
Basic actions ... 254
Naming rules ... 255
Service quota basics ... 256
More information ... 258
Core components ... 258
Tables, items, and attributes ... 259
Primary key ... 262
Secondary indexes ... 264
DynamoDB Streams ... 267
DynamoDB API ... 269
Control plane ... 269
Data plane ... 270
DynamoDB Streams ... 271

Transactions ... 272
Supported data types and naming rules ... 272
Naming rules ... 273
Data types ... 274
Data type descriptors ... 279
DynamoDB table classes ... 279
Partitions and data distribution in DynamoDB ... 280
Data distribution: Partition key ... 280
Data distribution: Partition key and sort key ... 282
Learn how to go from SQL to NoSQL ... 284
Relational or NoSQL? ... 285
Accessing and authentication ... 288
Creating a table ... 291
Getting information about a table ... 293
Writing data to a table ... 295
Reading data from a table ... 299
Managing indexes ... 308
Modifying data in a table ... 314
Deleting data from a table ... 317
Removing a table ... 320
Amazon DynamoDB learning resources and tools ... 320
Tools for coding and visualization ... 321
Prescriptive Guidance ... 321
Knowledge Center ... 322
Blog posts, repositories, and guides ... 323
Data modeling and design patterns ... 324
Training courses ... 324
Reads and writes ... 326
DynamoDB read consistency ... 326
Eventually consistent reads ... 326
Strongly consistent reads ... 327
Global tables read consistency ... 327
Read and write operations ... 327
Read operation consumption ... 327
Write operation consumption ... 329
DynamoDB throughput capacity ... 331

On-demand mode ... 331
Provisioned mode ... 332
DynamoDB on-demand capacity mode ... 332
Read request units and write request units ... 333
Initial throughput and scaling properties ... 334
DynamoDB maximum throughput for on-demand tables ... 334
Provisioned capacity mode ... 337
Read and write capacity units ... 338
Choosing initial throughput settings ... 338
DynamoDB auto scaling ... 339
Managing throughput capacity with auto scaling ... 340
Reserved capacity ... 371
Warm throughput ... 372
Check your table's warm throughput ... 373
Increase your table's warm throughput ... 376
Create a table with higher warm throughput ... 383
Warm throughput scenarios ... 394
Burst and adaptive capacity ... 396
Burst capacity ... 396
Adaptive capacity ... 396
Switching capacity modes ... 398
Provisioned mode to on-demand mode ... 398
On-demand mode to provisioned mode ... 400
Programming with DynamoDB ... 402
Overview of AWS SDK support for DynamoDB ... 402
SDK support of AWS account-based endpoints ... 404
Programmatic interfaces that work with DynamoDB ... 405
Higher-level programming interfaces ... 412
Running the code examples ... 473
Low-level API ... 480
Programming with Python ... 486
About Boto ... 487
Boto documentation ... 487
Client and resource layers ... 488
Using batch_writer ... 491
Additional code examples ... 492

Sessions and thread safety ... 492
Config ... 493
Error handling ... 497
Logging ... 499
Event hooks ... 501
Pagination and the Paginator ... 502
Waiters ... 504
Programming with JavaScript ... 504
About AWS SDK for JavaScript ... 505
AWS SDK for JavaScript V3 ... 505
JavaScript documentation ... 505
Abstraction layers ... 506
Marshall utility function ... 508
Reading items ... 509
Conditional writes ... 510
Pagination ... 511
Config ... 513
Waiters ... 516
Error handling ... 517
Logging ... 519
Considerations ... 520
Programming with the AWS SDK for Java 2.x ... 521
About the AWS SDK for Java 2.x ... 521
Getting started ... 522
SDK for Java 2.x documentation ... 532
Supported interfaces ... 532
Additional code examples ... 547
Sync and async programming ... 547
HTTP clients ... 548
Config ... 549
Error handling ... 556
AWS request ID ... 557
Logging ... 557
Pagination ... 560
Data class annotations ... 561
Error handling ... 561

Error components ... 562
Transactional errors ... 563
Error messages and codes ... 563
Error handling in your application ... 568
Error retries and exponential backoff ... 568
Batch operations and error handling ... 569
Working with AWS SDKs ... 570
Working with DynamoDB ... 572
Working with tables ... 572
Basic operations on tables ... 573
Considerations when choosing a table class in DynamoDB ... 582
Tags and labels ... 583
Working with global tables ... 588
Consistency modes ... 589
Account configurations ... 589
Core concepts ... 591
Same-account global table ... 595
Multi-account global tables ... 655
Global tables billing ... 677
Global tables versions ... 680
Global tables best practices ... 692
Working with items ... 693
Item sizes and formats ... 695
Reading an item ... 696
Writing an item ... 697
Return values ... 699
Batch operations ... 701
Atomic counters ... 703
Conditional writes ... 704
Using expressions ... 709
Time to Live (TTL) ... 750
Querying tables ... 780
Scanning tables ... 789
PartiQL query language ... 797
Working with items: Java ... 846
Working with items: .NET ... 877

Working with indexes ... 910
Global secondary indexes ... 915
Local secondary indexes ... 1010
Working with transactions ... 1062
How it works ... 1063
Using IAM with transactions ... 1072
Example code ... 1075
Working with streams ... 1079
Options ... 1080
Working with Kinesis Data Streams ... 1082
Working with DynamoDB Streams ... 1099
In-memory acceleration with DAX ... 1181
Use cases for DAX ... 1182
DAX usage notes ... 1183
How it works ... 1184
How DAX processes requests ... 1186
Item cache ... 1187
Query cache ... 1188
DAX cluster components ... 1189
Nodes ... 1190
Clusters ... 1190
Regions and availability zones ... 1191
Parameter groups ... 1192
Security groups ... 1192
Cluster ARN ... 1192
Cluster endpoint ... 1193
Node endpoints ... 1193
Subnet groups ... 1193
Events ... 1194
Maintenance window ... 1194
Creating a DAX cluster ... 1195
Creating an IAM service role for DAX to access DynamoDB ... 1196
DAX and IPv6 ... 1198
Using the AWS CLI ... 1200
Using the console ... 1206
Consistency models ... 1211

Consistency among DAX cluster nodes ... 1211
DAX item cache behavior ... 1212
DAX query cache behavior ... 1215
Strongly consistent and transactional reads ... 1216
Negative caching ... 1217
Strategies for writes ... 1217
Developing with the DAX client ... 1221
Tutorial: Running a sample application ... 1221
Modifying an existing application to use DAX ... 1285
Managing DAX clusters ... 1286
IAM permissions for managing a DAX cluster ... 1287
Scaling a DAX cluster ... 1290
Customizing DAX cluster settings ... 1291
Configuring TTL settings ... 1292
Tagging support for DAX ... 1294
AWS CloudTrail integration ... 1295
Deleting a DAX cluster ... 1295
Monitoring DynamoDB Accelerator ... 1295
DAX monitoring tools ... 1296
Monitoring with CloudWatch ... 1298
Logging DAX operations using AWS CloudTrail ... 1323
DAX T3/T2 burstable instances ... 1323
DAX T2 instance family ... 1324
DAX T3 instance family ... 1324
DAX access control ... 1325
IAM service role for DAX ... 1326
IAM policy to allow DAX cluster access ... 1328
Case study: Accessing DynamoDB and DAX ... 1329
Access to DynamoDB, but no access with DAX ... 1330
Access to DynamoDB and to DAX ... 1332
Access to DynamoDB via DAX, but no direct access to DynamoDB ... 1338
DAX encryption at rest ... 1340
Enabling encryption at rest using the AWS Management Console ... 1343
DAX encryption in transit ... 1343
Using service-linked roles for DAX ... 1344
Service-linked role permissions for DAX ... 1345

Creating a service-linked role for DAX ... 1346
Editing a service-linked role for DAX ... 1347
Deleting a service-linked role for DAX ... 1347
Accessing DAX across AWS accounts ... 1348
Set up IAM ... 1349
Set up a VPC ... 1352
Modify the DAX client to allow cross-account access ... 1354
DAX cluster sizing guide ... 1359
Overview ... 1359
Estimating traffic ... 1360
Load testing ... 1361
Data modeling ... 1363
Working with Item Collections ... 1364
Speed up queries by organizing your data with item collections ... 1366
Data modeling foundations ... 1366
Single table design ... 1366
Multiple table design ... 1369
Data modeling building blocks ... 1370
Composite sort key ... 1371
Multi-tenancy ... 1372
Sparse index ... 1373
Time to live ... 1375
Time to live archival ... 1376
Vertical partitioning ... 1377
Write sharding ... 1380
Data modeling schema design packages ... 1381
Prerequisites ... 1382
Social network ... 1383
Gaming profile ... 1392
Complaint management system ... 1401
Recurring payments ... 1418
Device status updates ... 1423
Online shop ... 1436
Relational modeling ... 1460
Traditional relational database models ... 1460
How DynamoDB eliminates the need for JOIN operations ... 1463

How DynamoDB transactions eliminate overhead to the write process ... 1464
First steps ... 1465
Example ... 1467
Migrating to DynamoDB ... 1479
Reasons to migrate ... 1479
Considerations when migrating ... 1480
How it works ... 1482
Migration tools ... 1483
Choosing a migration strategy ... 1484
Offline migration ... 1486
Hybrid migration ... 1488
Online - migrating each table 1:1 ... 1489
Online - migrating with a custom staging table ... 1491
NoSQL Workbench ... 1493
Download ... 1494
Data modeler ... 1495
Creating a new model ... 1495
Importing an existing model ... 1497
Editing an existing model ... 1498
Adding sample data ... 1498
Adding and validating access patterns ... 1499
Importing from CSV ... 1500
Facets ... 1501
Aggregate view ... 1501
Exporting a model ... 1502
Committing a data model ... 1503
Operation builder ... 1504
Connecting to datasets ... 1504
Building operations ... 1506
Cloning tables ... 1517
Exporting to CSV ... 1519
Sample data models ... 1519
Employee data model ... 1520
Discussion forum data model ... 1520
Music library data model ... 1521
Ski resort data model ... 1521

Credit card offers data model ... 1522
Bookmarks data model ... 1522
Release history ... 1523
Backup and restore ... 1529
Point-in-time backups ... 1530
Before you begin ... 1531
Enable point-in-time recovery ... 1532
On-demand backups ... 1536
How it works ... 1536
Backing up a table ... 1539
Restoring a table ... 1542
Deleting a table backup ... 1548
Using IAM ... 1549
Billing for backups ... 1557
How it works ... 1557
DynamoDB backup billing example ... 1558
Restores ... 1562
Restoring a table using point-in-time recovery ... 1562
Restoring a DynamoDB table to a point in time ... 1564
Using AWS Backup ... 1570
How it works ... 1571
Creating backups ... 1574
Copying backups ... 1576
Restoring a table ... 1577
Deleting backups ... 1578
On-demand backups managed by AWS Backup versus DynamoDB ... 1578
Code examples ... 1580
Basics ... 1583
Hello DynamoDB ... 1584
Learn the basics ... 1594
Actions ... 1751
Scenarios ... 2154
Accelerate reads with DAX ... 2156
Advanced Global Secondary Index scenarios ... 2165
Build an app to submit data to a DynamoDB table ... 2168
Compare multiple values with a single attribute ... 2169

Conditionally update an item's TTL ... 2192
Connect to a local instance ... 2199
Count expression operators ... 2200
Create a REST API to track COVID-19 data ... 2232
Create a messenger application ... 2233
Create a serverless application to manage photos ... 2234
Create a table with global secondary index ... 2238
Create a table with warm throughput enabled ... 2243
Create a web application to track DynamoDB data ... 2253
Create a websocket chat application ... 2254
Create an item with a TTL ... 2255
Create and manage MRSC global tables ... 2262
Create and manage global tables demonstrating MREC ... 2282
Delete data using PartiQL DELETE ... 2291
Detect PPE in images ... 2298
Insert data using PartiQL INSERT ... 2298
Invoke a Lambda function from a browser ... 2303
Manage Global Secondary Indexes ... 2304
Manage resource-based policies ... 2306
Monitor DynamoDB performance ... 2309
Perform advanced query operations ... 2309
Perform list operations ... 2344
Perform map operations ... 2377
Perform set operations ... 2401
Query a table by using batches of PartiQL statements ... 2431
Query a table using PartiQL ... 2493
Query a table using a Global Secondary Index ... 2548
Query a table using a begins_with condition ... 2556
Query a table using a date range ... 2562
Query a table with a complex filter expression ... 2571
Query a table with a dynamic filter expression ... 2579
Query a table with a filter expression and limit ... 2588
Query a table with nested attributes ... 2592
Query a table with pagination ... 2602
Query a table with strongly consistent reads ... 2612
Query data using PartiQL SELECT ... 2617

Query for TTL items ... 2622
Query tables using date and time patterns ... 2626
Save EXIF and other image information ... 2641
Set up Attribute-Based Access Control ... 2641
Understand update expression order ... 2645
Update a table's warm throughput setting ... 2676
Update an item's TTL ... 2681
Update data using PartiQL UPDATE ... 2685
Use API Gateway to invoke a Lambda function ... 2692
Use Step Functions to invoke Lambda functions ... 2694
Use a document model ... 2694
Use a high-level object persistence model ... 2710
Use atomic counter operations ... 2720
Use conditional operations ... 2743
Use expression attribute names ... 2773
Use scheduled events to invoke a Lambda function ... 2800
Work with Local Secondary Indexes ... 2802
Work with Streams and Time-to-Live ... 2805
Work with global tables and multi-Region replication eventual consistency (MREC) ... 2808
Work with resource tagging ... 2812
Work with table encryption ... 2815
Serverless examples ... 2817
Invoke a Lambda function from a DynamoDB trigger ... 2817
Reporting batch item failures for Lambda functions with a DynamoDB trigger ... 2826
AWS community contributions ... 2837
Build and test a serverless application ... 2838
Security ... 2840
AWS managed policies ... 2841
AWS managed policies ... 2841
AWS managed policy: AmazonDynamoDBFullAccess_v2 ... 2842
AmazonDynamoDBReadOnlyAccess ... 2842
DynamoDB updates to AWS managed policies ... 2843
Resource-based policies ... 2845
Create table ... 2846
Attach resource-based policy ... 2852
Attach policy to a stream ... 2857

Remove resource-based policy ... 2860
Cross-account access ... 2860
Blocking public access ... 2863
API operations ... 2866
IAM authorization ... 2874
Examples ... 2874
Considerations ... 2882
Best practices ... 2883
Attribute-based access control ... 2884
Why should I use ABAC? ... 2886
Condition keys ... 2887
Considerations ... 2887
Enabling ABAC in DynamoDB ... 2887
Using ABAC ... 2890
Example use cases ... 2892
Troubleshooting ... 2906
Data protection ... 2907
Encryption at rest ... 2908
Securing DynamoDB connections ... 2934
IAM ... 2936
Identity and Access Management ... 2936
Using conditions ... 2966
Compliance validation ... 2993
Resilience ... 2994
Infrastructure security ... 2995
Using VPC endpoints ... 2995
AWS PrivateLink for DynamoDB ... 3004
Types of Amazon VPC endpoints ... 3005
Considerations when using AWS PrivateLink for Amazon DynamoDB ... 3006
Creating an Amazon VPC endpoint ... 3007
Accessing Amazon DynamoDB interface endpoints ... 3007
Accessing DynamoDB tables and control API operations from DynamoDB interface endpoints ... 3008
Updating an on-premises DNS configuration ... 3010
Creating an Amazon VPC endpoint policy ... 3012
Using DynamoDB endpoints with AWS Management Console Private Access ... 3013

AWS PrivateLink for DynamoDB Streams ... 3014
AWS PrivateLink for DAX ... 3020
Configuration and vulnerability analysis ... 3023
Security best practices ... 3023
Preventative security best practices ... 3024
Detective security best practices ... 3026
Monitoring and logging ... 3030
Monitoring plan ... 3030
Performance baseline ... 3030
Integrated services ... 3031
Automated monitoring tools ... 3031
Monitoring metrics ... 3032
How do I use DynamoDB metrics? ... 3032
Viewing metrics in the CloudWatch console ... 3033
Viewing metrics in the AWS CLI ... 3034
Metrics and dimensions ... 3035
Creating CloudWatch alarms in DynamoDB ... 3065
Logging operations ... 3069
DynamoDB information in CloudTrail ... 3069
Understanding DynamoDB log file entries ... 3073
Contributor Insights ... 3092
How it works ... 3093
Getting started ... 3103
Using IAM ... 3113
Best practices ... 3120
NoSQL design ... 3120
NoSQL vs. RDBMS ... 3121
Two key concepts ... 3121
General approach ... 3122
NoSQL Workbench ... 3123
The DynamoDB Well-Architected Lens ... 3123
Cost optimization ... 3123
Conducting the Amazon DynamoDB Well-Architected Lens review ... 3173
The pillars of the Amazon DynamoDB Well-Architected Lens ... 3173
Partition key design ... 3175
Distributing workloads ... 3176

Write sharding ... 3178
Uploading data efficiently ... 3179
Sort key design ... 3181
Version control ... 3181
Secondary indexes ... 3183
General guidelines ... 3184
Sparse indexes ... 3187
Aggregation ... 3188
GSI overloading ... 3195
GSI sharding ... 3197
Creating a replica ... 3202
Large items ... 3203
Compression ... 3203
Vertical partitioning ... 3203
Using Amazon S3 ... 3204
Time series data ... 3205
Design pattern for time series data ... 3205
Time series table examples ... 3205
Many-to-many relationships ... 3206
Adjacency lists ... 3207
Materialized graphs ... 3208
Querying and scanning ... 3213
Scan performance ... 3213
Avoid spikes ... 3213
Parallel scans ... 3216
Table design ... 3218
Using global tables ... 3218
Key facts ... 3219
Key facts about MREC ... 3220
Key facts about MRSC ... 3221
Use cases ... 3223
Write modes ... 3224
Routing strategies in DynamoDB ... 3230
Evacuation processes ... 3236
Throughput capacity planning ... 3239
Preparation checklist ... 3240

Conclusion and resources ... 3248
Control plane ... 3249
Bulk data operations ... 3250
Conditional batch update ... 3250
Efficient bulk operations ... 3255
Handling concurrent updates ... 3256
Optimistic locking ... 3258
Pessimistic locking with transactions ... 3260
Distributed locking ... 3263
Choosing a concurrency control strategy ... 3266
Billing and Usage Reports ... 3267
Throughput Capacity ... 3270
Streams ... 3274
Storage ... 3275
Backup and Restore ... 3276
Data Transfer ... 3279
CloudWatch ... 3280
DAX ... 3281
Migrating a DynamoDB table from one account to another ... 3282
Migrate a table using AWS Backup for cross-account backup and restore ... 3283
Migrate a table using export to S3 and import from S3 ... 3285
DAX prescriptive guidance ... 3288
Evaluating the suitability of DAX ... 3288
Configuring your DAX client ... 3291
Configuring your DAX cluster ... 3292
Sizing your DAX cluster ... 3298
Deploying a cluster ... 3304
Cluster operations ... 3306
Monitoring DAX ... 3309
Using DynamoDB with other AWS services ... 3312
Integrating with Amazon Cognito ... 3312
Integrating with Amazon Redshift ... 3315
Cross-account integration considerations with CMK ... 3315
Zero-ETL integration with Amazon Redshift ... 3319
Loading data from DynamoDB into Amazon Redshift with COPY ... 3328
Integrating with Amazon EMR ... 3329

Overview ... 3330
Tutorial: Working with Amazon DynamoDB and Apache Hive ... 3331
Creating an external table in Hive ... 3339
Processing HiveQL statements ... 3343
Querying data in DynamoDB ... 3344
Copying data to and from Amazon DynamoDB ... 3346
Performance tuning ... 3360
Integrating with S3 ... 3366
Import from Amazon S3 ... 3366
Export to Amazon S3 ... 3386
Integrating with Amazon SageMaker Lakehouse ... 3413
Zero-ETL integration with Amazon SageMaker Lakehouse ... 3414
Integrating with Amazon OpenSearch Service ... 3415
How it works ... 3416
Creating an integration ... 3417
Next steps ... 3417
Handling breaking changes ... 3417
Zero-ETL integration with OpenSearch Service ... 3421
Integrating with Amazon EventBridge ... 3424
How it works ... 3424
Creating an integration through the console ... 3425
Next steps ... 3428
Integrating with Amazon MSK ... 3428
How it works ... 3429
Example integration ... 3429
Next steps ... 3436
Integration best practices ... 3436
Creating a snapshot ... 3436
Change data capture ... 3437
Using agentic AI with DynamoDB ... 3438
Generative AI use cases for DynamoDB ... 3438
Generative AI blogs for DynamoDB ... 3439
Leveraging DynamoDB Zero-ETL integration with OpenSearch Service ... 3439
LangGraph checkpoint store ... 3440
Key features ... 3440
Prerequisites ... 3441

Installation ... 3441
Basic usage ... 3441
Production configuration ... 3442
DynamoDB table configuration ... 3443
Required IAM permissions ... 3444
Asynchronous usage ... 3445
Clean up ... 3446
Error handling ... 3446
Additional resources ... 3446
Quotas and constraints ... 3448
Performing quota management tasks ... 3448
Accessing DynamoDB quotas ... 3448
Viewing current quotas in the console ... 3449
Viewing current quotas using the AWS CLI ... 3449
Requesting a quota increase ... 3451
Quotas ... 3452
Read/write throughput ... 3453
Reserved Capacity ... 371
Tables ... 3456
Global tables ... 3456
Secondary indexes ... 3457
Projected secondary index attributes ... 3458
DynamoDB Streams ... 3458
Import from Amazon S3 ... 3458
Table export to Amazon S3 ... 3459
Backup and restore ... 3459
Contributor Insights ... 3459
Constraints ... 3459
Read/write capacity mode ... 3460
Secondary indexes ... 3457
Partition keys and sort keys ... 3461
Naming rules ... 3462
Data types ... 3463
Items ... 3463
Attributes ... 3464
Expression parameters ... 3464

DynamoDB transactions ... 3465
DynamoDB Streams ... 3466
DynamoDB Accelerator (DAX) ... 3466
API-specific constraints ... 3467
DynamoDB encryption at rest ... 3470
API reference ... 3471
Troubleshooting ... 3472
Internal server errors ... 3472
Investigating internal server errors ... 3472
Minimizing the impact from internal server errors ... 3473
Improving operational awareness ... 3474
Latency ... 3478
Throttling ... 3481
Diagnosing throttling ... 3483
Resolution guide ... 3489
GSI write throttling and back-pressure ... 3514
CloudWatch throttling metrics ... 3516
Appendix ... 3519
Troubleshooting SSL/TLS connection establishment issues with DynamoDB ... 3519
Testing your application or service ... 3519
Testing your client browser ... 3520
Updating your software application client ... 3520
Updating your client browser ... 3521
Manually updating your certificate bundle ... 3521
Example tables and data for use in DynamoDB ... 3522
Sample data files ... 3523
Creating example tables and uploading data ... 3536
Creating example tables and uploading data - Java ... 3536
Creating example tables and uploading data - .NET ... 3546
Example application using AWS SDK for Python (Boto3) ... 3558
Step 1: Deploy and test locally ... 3559
Step 2: Examine the data model and implementation details ... 3564
Step 3: Deploy in production ... 3574
Step 4: Clean up resources ... 3583
Reserved words in DynamoDB ... 3584
AWS SDK for Java 1.x examples ... 3597

DAX and Java SDK v1 ... 3597
Modifying an existing SDK for Java 1.x application to use DAX ... 3609
Querying global secondary indexes with SDK for Java 1.x ... 3614
AWS SDK for Go 1.x examples ... 3618
Go and DAX ... 3618
AWS SDK for Node.js 2.x examples ... 3621
Node.js and DAX ... 3621
Document history ... 3632
Earlier updates ... 3667
Legacy features ... 3699
Global tables version 2017.11.29 (Legacy) ... 3699
How it works ... 3700
Best Practices and Requirements ... 3705
Creating a global table (Version 2017.11.29) ... 3709
Monitoring global tables ... 3713
Using IAM with global tables ... 3714
Previous low-level DynamoDB API version (2011-12-05) ... 3718
BatchGetItem ... 3719
BatchWriteItem ... 3726
CreateTable ... 3733
DeleteItem ... 3741
DeleteTable ... 3748
DescribeTables ... 3752
GetItem ... 3756
ListTables ... 3760
PutItem ... 3763
Query ... 3770
Scan ... 3784
UpdateItem ... 3802
UpdateTable ... 3812
Legacy DynamoDB conditional parameters ... 3817
AttributesToGet ... 3819
AttributeUpdates ... 3820
ConditionalOperator ... 3822
Expected ... 3823
KeyConditions ... 3829

QueryFilter 3832
ScanFilter 3834
Writing conditions with legacy parameters 3836

What is Amazon DynamoDB?

Amazon DynamoDB is a serverless, fully managed, distributed NoSQL database with single-digit millisecond performance at any scale.

DynamoDB addresses your needs to overcome scaling and operational complexities of relational databases. DynamoDB is purpose-built and optimized for operational workloads that require consistent performance at any scale. For example, DynamoDB delivers consistent single-digit millisecond performance for a shopping cart use case, whether you have 10 or 100 million users. Launched in 2012, DynamoDB continues to help you move away from relational databases while reducing cost and improving performance at scale.

Customers across all sizes, industries, and geographies use DynamoDB to build modern, serverless applications that can start small and scale globally. DynamoDB scales to support tables of virtually any size while providing consistent single-digit millisecond performance and high availability.

For events, such as Amazon Prime Day, DynamoDB powers multiple high-traffic Amazon properties and systems, including Alexa, Amazon.com sites, and all Amazon fulfillment centers. For such events, DynamoDB APIs have handled trillions of calls from Amazon properties and systems. DynamoDB continuously serves hundreds of customers with tables that have peak traffic of over half a million requests per second. It also serves hundreds of customers whose table sizes exceed 200 TB, and processes over one billion requests per hour.

Topics

- Characteristics of DynamoDB
- DynamoDB use cases
- Capabilities of DynamoDB
- Service integrations
- Security
- Resilience
- Accessing DynamoDB
- DynamoDB pricing
- Getting started with DynamoDB

Characteristics of DynamoDB

Serverless

With DynamoDB, you don't need to provision any servers, or patch, manage, install, maintain, or operate any software. DynamoDB provides zero downtime maintenance. It has no versions (major, minor, or patch), and there are no maintenance windows.

DynamoDB's on-demand capacity mode offers pay-as-you-go pricing for read and write requests so you only pay for what you use. With on-demand, DynamoDB instantly scales up or down your tables to adjust for capacity and maintains performance with zero administration. It also scales down to zero so you don't pay for throughput when your table doesn't have traffic and there are no cold starts.

NoSQL

As a NoSQL database, DynamoDB is purpose-built to deliver improved performance, scalability, manageability, and flexibility compared to traditional relational databases. To support a wide variety of use cases, DynamoDB supports both key-value and document data models.

Unlike relational databases, DynamoDB doesn't support a JOIN operator. We recommend that you denormalize your data model to reduce database round trips and processing power needed to answer queries. As a NoSQL database, DynamoDB provides strong read consistency and ACID transactions to build enterprise-grade applications.

Fully managed

As a fully managed database service, DynamoDB handles the undifferentiated heavy lifting of managing a database so that you can focus on building value for your customers. It handles setup, configurations, maintenance, high availability, hardware provisioning, security, backups, monitoring, and more. This ensures that when you create a DynamoDB table, it's instantly ready for production workloads. DynamoDB constantly improves its availability, reliability, performance, security, and functionality without requiring upgrades or downtime.

Single-digit millisecond performance at any scale

DynamoDB was purpose-built to improve upon the performance and scalability of relational databases to deliver single-digit millisecond performance at any scale. To achieve this scale and

performance, DynamoDB is optimized for high-performance workloads and provides APIs that encourage efficient database usage. It omits features that are inefficient and non-performing at scale, for example, JOIN operations. DynamoDB delivers consistent single-digit millisecond performance for your application, whether you have 100 or 100 million users.

DynamoDB use cases

Customers across all sizes, industries, and geographies use DynamoDB to build modern, serverless applications that can start small and scale globally. DynamoDB is ideal for use cases that require consistent performance at any scale with little to zero operational overhead. The following list presents some use cases where you can use DynamoDB:

- **Financial service applications** – Suppose you're a financial services company building applications, such as live trading and routing, loan management, token generation, and transaction ledgers. With DynamoDB global tables, your applications can respond to events and serve traffic from your chosen AWS Regions with fast, local read and write performance.

  DynamoDB is suitable for applications with the most stringent availability requirements. It removes the operational burden of manually scaling instances for increased storage or throughput, versioning, and licensing.

  You can use DynamoDB transactions to achieve atomicity, consistency, isolation, and durability (ACID) across one or more tables with a single request. (ACID) transactions suit workloads that include processing financial transactions or fulfilling orders. DynamoDB instantly accommodates your workloads as they ramp up or down, enabling you to efficiently scale your database for market conditions, such as trading hours.
- **Gaming applications** – As a gaming company, you can use DynamoDB for all parts of game platforms, for example, game state, player data, session history, and leaderboards. Choose DynamoDB for its scale, consistent performance, and the ease of operations provided by its serverless architecture. DynamoDB is well suited for scale-out architectures needed to support successful games. It quickly scales your game's throughput both in and out (scale to zero with no cold start). This scalability optimizes your architecture's efficiency whether you're scaling out for peak traffic or scaling back when gameplay usage is low.
- **Streaming applications** – Media and entertainment companies use DynamoDB as a metadata index for content, content management service, or to serve near real-time sports statistics. They also use DynamoDB to run user watchlist and bookmarking services and process billions of daily customer events for generating recommendations. These customers benefit from DynamoDB's

scalability, performance, and resiliency. DynamoDB scales to workload changes as they ramp up or down, enabling streaming media use cases that can support any levels of demand.

To learn more about how customers from different industries use DynamoDB, see Amazon DynamoDB Customers and This is My Architecture.

Capabilities of DynamoDB

Multi-active replication with global tables

Global tables provide multi-active replication of your data across your chosen AWS Regions with 99.999% availability. Global tables deliver a fully managed solution for deploying a multi-Region, multi-active database, without building and maintaining your own replication solution. With global tables, you can specify the AWS Regions where you want the tables to be available. DynamoDB replicates ongoing data changes to all of these tables.

Your globally distributed applications can access data locally in your selected Regions to achieve single-digit millisecond read and write performance. Because global tables are multi-active, you don't need a primary table. This means there are no complicated or delayed fail-overs, or database downtime when failing over an application between Regions.

ACID transactions

DynamoDB is built for mission-critical workloads. It includes (ACID) transactions support for applications that require complex business logic. DynamoDB provides native, server-side support for transactions, simplifying the developer experience of making coordinated, all-or-nothing changes to multiple items within and across tables.

Change data capture for event-driven architectures

DynamoDB supports streaming of item-level change data capture (CDC) records in near-real time. It offers two streaming models for CDC: DynamoDB Streams and Kinesis Data Streams for DynamoDB. Whenever an application creates, updates, or deletes items in a table, streams records a time-ordered sequence of every item-level change in near-real time. This makes DynamoDB Streams ideal for applications with event-driven architecture to consume and act upon the changes.

Secondary indexes

DynamoDB offers the option to create both global and local secondary indexes, which let you query the table data using an alternate key. With these secondary indexes, you can access data with attributes other than the primary key, giving you maximum flexibility in accessing your data.

Service integrations

DynamoDB broadly integrates with several AWS services to help you get more value from your data, eliminate undifferentiated heavy lifting, and operate your workloads at scale. Some examples are: AWS CloudFormation, Amazon CloudWatch, Amazon S3, AWS Identity and Access Management (IAM), and AWS Auto Scaling. The following sections describe some of the service integrations that you can perform using DynamoDB:

Serverless integrations

To build end-to-end serverless applications, DynamoDB integrates natively with a number of serverless AWS services. For example, you can integrate DynamoDB with AWS Lambda to create triggers, which are pieces of code that automatically respond to events in DynamoDB Streams. With triggers, you can build event-driven applications that react to data modifications in DynamoDB tables. For cost optimization, you can filter events that Lambda processes from a DynamoDB stream.

The following list presents some examples of serverless integrations with DynamoDB:

- AWS AppSync for creating GraphQL APIs
- Amazon API Gateway for creating REST APIs
- Lambda for serverless compute
- Amazon Kinesis Data Streams for change data capture (CDC)

Importing and exporting data to Amazon S3

Integrating DynamoDB with Amazon S3 enables you to easily export data to an Amazon S3 bucket for analytics and machine learning. DynamoDB supports full table exports and incremental exports to export changed, updated, or deleted data between a specified time period. You can also import data from Amazon S3 into a new DynamoDB table.

Zero-ETL integration

DynamoDB supports zero-ETL integration with Amazon Redshift and Using an OpenSearch Ingestion pipeline with Amazon DynamoDB. These integrations enable you to run complex analytics and use advanced search capabilities on your DynamoDB table data. For example, you can perform full-text and vector search, and semantic search on your DynamoDB data. Zero-ETL integrations have no impact on production workloads running on DynamoDB.

Caching

DynamoDB Accelerator (DAX) is a fully managed, highly available caching service built for DynamoDB. DAX delivers up to 10 times performance improvement – from milliseconds to microseconds – even at millions of requests per second. DAX does all the heavy lifting required to add in-memory acceleration to your DynamoDB tables, without requiring you to manage cache invalidation, data population, or cluster management.

Security

DynamoDB utilizes IAM to help you securely control access to your DynamoDB resources. With IAM, you can centrally manage permissions that control which DynamoDB users can access resources. You use IAM to control who is authenticated (signed in) and authorized (has permissions) to use resources. Because DynamoDB utilizes IAM, there are no user names or passwords for accessing DynamoDB. Because you don't have any complicated password rotation policies to manage, it simplifies your security posture. With IAM, you can also enable fine-grained access control to provide authorization at the attribute level. You can also define resource-based policies with support for IAM Access Analyzer and Block Public Access (BPA) to simplify policy management.

By default, DynamoDB encrypts all customer data at rest. Encryption at rest enhances the security of your data by using encryption keys stored in AWS Key Management Service (AWS KMS). With encryption at rest, you can build security-sensitive applications that meet strict encryption compliance and regulatory requirements. When you access an encrypted table, DynamoDB decrypts the table data transparently. You don't have to change any code or applications to use or manage encrypted tables. DynamoDB continues to deliver the same single-digit millisecond latency that you have come to expect, and all DynamoDB queries work seamlessly on your encrypted data.

You can specify whether DynamoDB should use an AWS owned key (default encryption type), AWS managed key, or a Customer managed key to encrypt user data. The default encryption using AWS-

owned KMS keys is available at no additional charge. For client-side encryption, you can use the AWS Database Encryption SDK.

DynamoDB also adheres to several compliance standards, including HIPAA, PCI DSS, and GDPR, which enables you to meet regulatory requirements.

Resilience

By default, DynamoDB automatically replicates your data across three Availability Zones to provide high durability and a 99.99% availability SLA. DynamoDB also provides additional capabilities to help you achieve your business continuity and disaster recovery objectives.

DynamoDB includes the following features to help support your data resiliency and backup needs:

Features

- Global tables
- Continuous backups and point-in-time recovery
- On-demand backup and restore

Global tables

DynamoDB global tables enable a 99.999% availability SLA and multi-Region resilience. This helps you build resilient applications and optimize them for the lowest recovery time objective (RTO) and recovery point objective (RPO). Global tables also integrates with AWS Fault Injection Service (AWS FIS) to perform fault injection experiments on your global table workloads. For example, pausing global table replication to any replica table.

Continuous backups and point-in-time recovery

Continuous backups provide you per-second granularity and the ability to initiate a point-in-time recovery. With point-in-time recovery, you can restore a table to any point in time up to the second during the last 35 days. You can set the recovery period to any value between 1 and 35 days.

Continuous backups and initiating a point-in-time restore doesn't use provisioned capacity. They also don't have any impact on the performance or availability of your applications.

On-demand backup and restore

On-demand backup and restore let you create full backups of a table for long-term retention and archival for regulatory compliance needs. Backups don't impact the performance of your table and you can back up tables of any size. With AWS Backup integration, you can use AWS Backup to schedule, copy, tag, and manage the life cycle of your DynamoDB on-demand backups automatically. Using AWS Backup, you can copy on-demand backups across accounts and Regions, and transition older backups to cold storage for cost-optimization.

Accessing DynamoDB

You can work with DynamoDB using the AWS Management Console, the AWS Command Line Interface, NoSQL Workbench for DynamoDB, or DynamoDB APIs.

For more information, see Accessing DynamoDB.

DynamoDB pricing

DynamoDB charges for reading, writing, and storing data in your tables, along with any optional features you choose to enable. DynamoDB has two capacity modes with their respective billing options for processing reads and writes on your tables: on-demand and provisioned.

DynamoDB is also included in the **always free tier**, providing 25 GB of storage. The **Always free tier** also includes 25 provisioned Write and 25 provisioned Read Capacity Units (WCU, RCU) which is enough to handle 200 M requests per month.

For more information, see Amazon DynamoDB pricing.

Getting started with DynamoDB

If you're a first-time user of DynamoDB, we recommend that you begin by reading the following topics:

- Getting started with DynamoDB – Walks you through the process of setting up DynamoDB, creating sample tables, and uploading data. This topic also provides information about performing some basic database operations using the AWS Management Console, AWS CLI, NoSQL Workbench, and DynamoDB APIs.
- DynamoDB core components – Describes the basic DynamoDB concepts.

- Best practices for designing and architecting with DynamoDB – Provides recommendations about NoSQL design, DynamoDB Well-Architected Lens, table design and several other DynamoDB features. These best practices help you maximize performance and minimize throughput costs when working with DynamoDB.

We also recommend that you review the following tutorials that present complete end-to-end procedures to familiarize yourself with DynamoDB. You can complete these tutorials using the **always free tier** feature.

- Create and Query a NoSQL Table with Amazon DynamoDB
- Build an Application Using a NoSQL Key-Value Data Store

For information about resources, tools, and strategies to migrate to DynamoDB, see Migrating to DynamoDB. To read the latest blogs and whitepapers, see Amazon DynamoDB resources.

Getting started with DynamoDB

You'll learn how to connect to, create, and manage DynamoDB tables in the following sections.

Before you begin, you should familiarize yourself with the basic concepts in Amazon DynamoDB. You can get a quick overview in What is Amazon DynamoDB? and a more in-depth look in Core components of Amazon DynamoDB. Then, continue on to Prerequisites.

> **Note**
>
> When you sign up for AWS, you can get started with DynamoDB using the AWS Free Tier. If you have not already exceeded the free tier benefits for Amazon DynamoDB, it won't cost you anything to complete the examples in this section. Otherwise, you'll incur the standard DynamoDB usage fees from the time that you create the tables until you delete the tables. If you don't want to sign up for a free tier account, you can set up DynamoDB local (downloadable version) on your computer. The downloadable version lets you develop and test applications locally without signing up for an AWS account or accessing the DynamoDB web service.

Topics

- Amazon DynamoDB resources for first-time users
- Accessing DynamoDB
- Prerequisites
- Setting up DynamoDB
- Step 1: Create a table in DynamoDB
- Step 2: Write data to a DynamoDB table
- Step 3: Read data from a DynamoDB table
- Step 4: Update data in a DynamoDB table
- Step 5: Query data in a DynamoDB table
- Step 6: (Optional) Delete your DynamoDB table to clean up resources
- Continue learning about DynamoDB
- Generate infrastructure code for Amazon DynamoDB using Console-to-Code

Amazon DynamoDB resources for first-time users

We recommend that first time users begin by reading the following sections, and refer to them as needed.

Example **Service highlights and pricing** Provides a general product overview of DynamoDB, common use cases, service highlights, and pricing.	**Example** **DynamoDB resources** Videos, tutorials, and prescriptive guidance that introduces you to the service, concepts of data modeling, and core features and capabilit ies.
Example **Getting started** Information on setting up DynamoDB, creating sample tables, and uploading data.	**Example** **DynamoDB basics course** Free digital course that teaches the fundament als of DynamoDB, including table design, data types, and basic operations.
Example **DynamoDB Nuggets** Short, focused video tutorials that explain key DynamoDB concepts and features.	**Example** **DynamoDB code samples repository** Practical DynamoDB code examples in various programming languages.
Example **Free DynamoDB training** AWS provides free digital training courses that cover DynamoDB concepts, features, and best practices.	**Example** **NoSQL Workbench for DynamoDB** Unified visual tool that provides data modeling, data visualization, and query development features.
Example **DynamoDB Design Patterns**	**Example** **Hands-on Tutorials**

Best practices and data modeling examples for different use cases with practical code samples.	Step-by-step tutorials in the AWS Management Console that guide you through common DynamoDB tasks.
Example **Migrating to DynamoDB** Overview of the process, tools, and strategies for migrating a database into DynamoDB.	**Example** **AWS Well-Architected Lens for DynamoDB** Architectural best practices for designing and operating reliable, secure, efficient, and cost-effective applications using DynamoDB.

Amazon DynamoDB additional best practices for first-time users

After you complete the preceding sections, read these sections:

- ***DynamoDB throughput capacity***

  Provides an overview of the two throughput modes available for DynamoDB and considerations in selecting the appropriate capacity mode for your application. On-demand mode is the default and recommended throughput option for most DynamoDB workloads.

- ***Best practices***

  Identify and address issues to maximize performance and minimize costs when working with DynamoDB.

AWS CLI resources

If you want to use the AWS Command Line Interface (AWS CLI), you can use these documents to help you get started:

- **AWS CLI documentation**

  This section provides information on downloading the AWS CLI, getting the AWS CLI working on your system, and providing your AWS credentials.

- **AWS CLI documentation for DynamoDB**

This separate document covers all of the AWS CLI for DynamoDB commands, including syntax and examples.

Programming resources

You can write application programs to use the DynamoDB API with a variety of popular programming languages. Here are some resources:

- **Tools for Amazon Web Services**

  AWS provides a number of software development kits (SDKs) with support for DynamoDB. You can code for DynamoDB using Java, .NET, PHP, Ruby, and other languages. These SDKs can greatly simplify your application development by formatting your requests to DynamoDB, parsing responses, and providing retry logic and error handling.

- **DynamoDB API Reference**

  If you don't want to use the AWS SDKs, you can interact with DynamoDB directly using the DynamoDB API. This document covers all of the DynamoDB API operations, including syntax and examples. You can find troubleshooting tips and information on creating and authenticating requests and handling responses in this section.

Accessing DynamoDB

You can access Amazon DynamoDB using the AWS Management Console, the AWS Command Line Interface (AWS CLI), or the DynamoDB API.

Topics

- Using the console
- Using the AWS CLI
- Using the API
- Using the NoSQL workbench for DynamoDB
- IP address ranges
- Dual-stack endpoints for Internet Protocol version 6 (IPv6)

Using the console

You can access the AWS Management Console for Amazon DynamoDB at https://console.aws.amazon.com/dynamodb/home.

Here are some of the actions you can perform in the DynamoDB console:

- **Manage tables**: Create, update, and delete tables. The capacity calculator can help estimate capacity requirements.
- **Interact with data**: View, add, update, and delete items in your tables. Manage Time to Live (TTL) settings.
- **Monitor and analyze**: View dashboards, monitor and set up alarms, and analyze metrics and alerts for your DynamoDB tables.
- **Optimize and extend**: Manage secondary indexes, streams, triggers, reserved capacity, and other advanced features to enhance your DynamoDB usage.

The DynamoDB console provides a comprehensive interface for managing your DynamoDB resources. We encourage you to access the console and interact with it to learn more.

Using the AWS CLI

You can use the AWS Command Line Interface (AWS CLI) to control multiple AWS services from the command line and automate them through scripts. You can use the AWS CLI for ad hoc operations, such as creating a table. You can also use it to embed Amazon DynamoDB operations within utility scripts.

Before you can use the AWS CLI with DynamoDB, you must get an access key ID and secret access key. For more information, see Granting programmatic access.

For a complete listing of all the commands available for DynamoDB in the AWS CLI, see the AWS CLI command reference.

Downloading and configuring the AWS CLI

The AWS CLI is available at http://aws.amazon.com/cli. It runs on Windows, macOS, or Linux. After you download the AWS CLI, follow these steps to install and configure it:

1. Go to the AWS Command Line Interface User Guide.
2. Follow the instructions for Installing the AWS CLI and Configuring the AWS CLI.

Using the AWS CLI with DynamoDB

The command line format consists of a DynamoDB operation name followed by the parameters for that operation. The AWS CLI supports a shorthand syntax for the parameter values, as well as JSON.

For example, the following command creates a table named *Music*. The partition key is *Artist*, and the sort key is *SongTitle*. (For easier readability, long commands in this section are broken into separate lines.)

```
aws dynamodb create-table \
    --table-name Music \
    --attribute-definitions \
        AttributeName=Artist,AttributeType=S \
        AttributeName=SongTitle,AttributeType=S \
    --key-schema AttributeName=Artist,KeyType=HASH
 AttributeName=SongTitle,KeyType=RANGE \
    --billing-mode PAY_PER_REQUEST \
    --table-class STANDARD
```

The following commands add new items to the table. These examples use a combination of shorthand syntax and JSON.

```
aws dynamodb put-item \
    --table-name Music \
    --item \
        '{"Artist": {"S": "No One You Know"}, "SongTitle": {"S": "Call Me Today"},
 "AlbumTitle": {"S": "Somewhat Famous"}}' \
    --return-consumed-capacity TOTAL

aws dynamodb put-item \
    --table-name Music \
    --item '{
        "Artist": {"S": "Acme Band"},
        "SongTitle": {"S": "Happy Day"},
        "AlbumTitle": {"S": "Songs About Life"} }' \
    --return-consumed-capacity TOTAL
```

On the command line, it can be difficult to compose valid JSON. However, the AWS CLI can read JSON files. For example, consider the following JSON code snippet, which is stored in a file named *key-conditions.json*.

```
{
    "Artist": {
        "AttributeValueList": [
            {
                "S": "No One You Know"
            }
        ],
        "ComparisonOperator": "EQ"
    },
    "SongTitle": {
        "AttributeValueList": [
            {
                "S": "Call Me Today"
            }
        ],
        "ComparisonOperator": "EQ"
    }
}
```

You can now issue a `Query` request using the AWS CLI. In this example, the contents of the *key-conditions.json* file are used for the `--key-conditions` parameter.

```
aws dynamodb query --table-name Music --key-conditions file://key-conditions.json
```

Using the AWS CLI with DynamoDB local

The AWS CLI can also interact with DynamoDB local (downloadable version) that runs on your computer. To enable this, add the following parameter to each command:

`--endpoint-url http://localhost:8000`

The following example uses the AWS CLI to list the tables in a local database.

```
aws dynamodb list-tables --endpoint-url http://localhost:8000
```

If DynamoDB is using a port number other than the default (8000), modify the `--endpoint-url` value accordingly.

> **Note**
>
> The AWS CLI can't use the DynamoDB local (downloadable version) as a default endpoint. Therefore, you must specify `--endpoint-url` with each command.

Using the API

You can use the AWS Management Console and the AWS Command Line Interface to work interactively with Amazon DynamoDB. However, to get the most out of DynamoDB, you can write application code using the AWS SDKs.

The AWS SDKs provide broad support for DynamoDB in Java, JavaScript in the browser, .NET, Node.js, PHP, Python, Ruby, C++, Go, Android, and iOS. .

Before you can use the AWS SDKs with DynamoDB, you must get an AWS access key ID and secret access key. For more information, see Setting up DynamoDB (web service).

For a high-level overview of DynamoDB application programming with the AWS SDKs, see Programming with DynamoDB and the AWS SDKs.

Using the NoSQL workbench for DynamoDB

You can also access DynamoDB by downloading and using the NoSQL Workbench for DynamoDB.

NoSQL Workbench for Amazon DynamoDB is a cross-platform, client-side GUI application that you can use for modern database development and operations. It's available for Windows, macOS, and Linux. NoSQL Workbench is a visual development tool that provides data modeling, data visualization, and query development features to help you design, create, query, and manage DynamoDB tables. NoSQL Workbench now includes DynamoDB local as an optional part of the installation process, which makes it easier to model your data in DynamoDB local. To learn more about DynamoDB local and its requirements, see Setting up DynamoDB local (downloadable version).

> **Note**
>
> The NoSQL Workbench for DynamoDB currently doesn't support AWS logins that are configured with two-factor authentication (2FA).

Data modeling

With NoSQL Workbench for DynamoDB, you can build new data models from, or design models based on, existing data models that satisfy your application's data access patterns. You can also import and export the designed data model at the end of the process. For more information, see Building data models with NoSQL Workbench.

Operation building

NoSQL Workbench provides a rich graphical user interface for you to develop and test queries. You can use the *operation builder* to view, explore, and query live datasets. You can also use the structured operation builder to build and perform data plane operations. It supports projection and condition expression, and lets you generate sample code in multiple languages. For more information, see Exploring datasets and building operations with NoSQL Workbench.

IP address ranges

Amazon Web Services (AWS) publishes its current IP address ranges in JSON format. To view the current ranges, download ip-ranges.json. For more information, see AWS IP address ranges in the AWS General Reference.

To find the IP address ranges that you can use to access to DynamoDB tables and indexes, search the ip-ranges.json file for the following string: `"service": "DYNAMODB"`.

> **Note**
>
> The IP address ranges do not apply to DynamoDB Streams or DynamoDB Accelerator (DAX).

Dual-stack endpoints for Internet Protocol version 6 (IPv6)

DynamoDB offers dual-stack endpoints that are compatible with both IPv4 and IPv6. The endpoint naming conventions are:

- `dynamodb.<region>.api.aws`
- `<account-id>.ddb.<region>.api.aws`
- `streams-dynamodb.<region>.api.aws`
- `dax.<region>.api.aws`
- `dynamodb-fips.<region>.api.aws`

For a complete list of DynamoDB endpoints and regional availability, see Amazon DynamoDB endpoints and quotas topic in the *AWS General Reference guide*.

For more information regarding setting up the AWS CLI to use dual-stack endpoints, see Set to use dual-stack endpoints for all AWS services section in the *AWS Command Line Interface guide*.

For more information regarding setting up your SDK clients to use dual-stack endpoints, see Dual-stack and FIPS endpoints topic in the *AWS SDKs and Tools guide*.

Before using DynamoDB with IPv6, you must update your IAM user role or resourced-based policies that you use for IP address filtering to include IPv6 address ranges. IP address filtering policies that do not account for IPv6 address can result in access issues. For more information, see IP address condition operators section in the *AWS Identity and Access Management guide*.

Prerequisites

Before starting the Amazon DynamoDB tutorial, learn about the ways you can access DynamoDB in Accessing DynamoDB. Then, set up DynamoDB through either the web service or the locally downloaded version in Setting up DynamoDB. After that, continue on to Step 1: Create a table in DynamoDB.

Note

- If you plan to interact with DynamoDB only through the AWS Management Console, you don't need an AWS access key. Complete the steps in Signing up for AWS, and then continue on to Step 1: Create a table in DynamoDB.
- If you don't want to sign up for a free tier account, you can set up DynamoDB local (downloadable version). Then continue on to Step 1: Create a table in DynamoDB.
- There are differences when working with CLI commands in terminals on Linux and Windows. The following guide presents commands formatted for Linux terminals (this includes macOS), and commands formatted for Windows CMD. Choose the command that best fits the terminal application you are using.

Setting up DynamoDB

In addition to the Amazon DynamoDB web service, AWS provides a downloadable version of DynamoDB that you can run on your computer. The downloadable version is helpful for developing

and testing your code. It lets you write and test applications locally without accessing the DynamoDB web service.

The topics in this section describe how to set up DynamoDB (downloadable version) and the DynamoDB web service.

Topics

- Setting up DynamoDB (web service)
- Setting up DynamoDB local (downloadable version)

Setting up DynamoDB (web service)

To use the Amazon DynamoDB web service:

1. Sign up for AWS.
2. Get an AWS access key (used to access DynamoDB programmatically).

> **Note**
>
> If you plan to interact with DynamoDB only through the AWS Management Console, you don't need an AWS access key, and you can skip ahead to Using the console.

3. Configure your credentials (used to access DynamoDB programmatically).

Signing up for AWS

To use the DynamoDB service, you must have an AWS account. If you don't already have an account, you are prompted to create one when you sign up. You're not charged for any AWS services that you sign up for unless you use them.

To sign up for AWS

1. Open https://portal.aws.amazon.com/billing/signup.
2. Follow the online instructions.

   Part of the sign-up procedure involves receiving a phone call or text message and entering a verification code on the phone keypad.

When you sign up for an AWS account, an *AWS account root user* is created. The root user has access to all AWS services and resources in the account. As a security best practice, assign administrative access to a user, and use only the root user to perform tasks that require root user access.

Granting programmatic access

Before you can access DynamoDB programmatically or through the AWS Command Line Interface (AWS CLI), you must have programmatic access. You don't need programmatic access if you plan to use the DynamoDB console only.

Users need programmatic access if they want to interact with AWS outside of the AWS Management Console. The way to grant programmatic access depends on the type of user that's accessing AWS.

To grant users programmatic access, choose one of the following options.

Which user needs programmatic access?	To	By
IAM	(Recommended) Use console credentials as temporary credentials to sign programmatic requests to the AWS CLI, AWS SDKs, or AWS APIs.	Following the instructions for the interface that you want to use. • For the AWS CLI, see Login for AWS local development in the *AWS Command Line Interface User Guide*. • For AWS SDKs, see Login for AWS local development in the *AWS SDKs and Tools Reference Guide*.
Workforce identity (Users managed in IAM Identity Center)	Use temporary credentials to sign programmatic requests to the AWS CLI, AWS SDKs, or AWS APIs.	Following the instructions for the interface that you want to use.

Which user needs programmatic access?	To	By
		• For the AWS CLI, see Configuring the AWS CLI to use AWS IAM Identity Center in the *AWS Command Line Interface User Guide*. • For AWS SDKs, tools, and AWS APIs, see IAM Identity Center authentication in the *AWS SDKs and Tools Reference Guide*.
IAM	Use temporary credentials to sign programmatic requests to the AWS CLI, AWS SDKs, or AWS APIs.	Following the instructions in Using temporary credentials with AWS resources in the *IAM User Guide*.

Which user needs programmatic access?	To	By
IAM	(Not recommended) Use long-term credentials to sign programmatic requests to the AWS CLI, AWS SDKs, or AWS APIs.	Following the instructions for the interface that you want to use. • For the AWS CLI, see Authenticating using IAM user credentials in the *AWS Command Line Interface User Guide*. • For AWS SDKs and tools, see Authenticate using long-term credentials in the *AWS SDKs and Tools Reference Guide*. • For AWS APIs, see Managing access keys for IAM users in the *IAM User Guide*.

Configuring your credentials

Before you can access DynamoDB programmatically or through the AWS CLI, you must configure your credentials to enable authorization for your applications.

There are several ways to do this. For example, you can manually create the credentials file to store your access key ID and secret access key. You also can use the AWS CLI command `aws configure` to automatically create the file. Alternatively, you can use environment variables. For more information about configuring your credentials, see the programming-specific AWS SDK developer guide.

To install and configure the AWS CLI, see Using the AWS CLI.

Integrating with other DynamoDB services

You can integrate DynamoDB with many other AWS services. For more information, see the following:

- Using DynamoDB with other AWS services
- CloudFormation for DynamoDB
- Using AWS Backup with DynamoDB
- AWS Identity and Access Management (IAM) and DynamoDB
- Using AWS Lambda with Amazon DynamoDB

Setting up DynamoDB local (downloadable version)

With the downloadable version of Amazon DynamoDB, you can develop and test applications without accessing the DynamoDB web service. Instead, the database is self-contained on your computer. When you're ready to deploy your application in production, you remove the local endpoint in the code, and then it points to the DynamoDB web service.

Having this local version helps you save on throughput, data storage, and data transfer fees. In addition, you don't need an internet connection while you develop your application.

DynamoDB local is available as a download (requires JRE), as an Apache Maven dependency, or as a Docker image.

If you prefer to use the Amazon DynamoDB web service instead, see Setting up DynamoDB (web service).

Topics

- Deploying DynamoDB locally on your computer
- DynamoDB local usage notes
- Release history for DynamoDB local
- Telemetry in DynamoDB local

Deploying DynamoDB locally on your computer

> **Note**
>
> - DynamoDB local is available in three versions: v3.x (Current), v2.x (Legacy), and v1.x (Deprecated).
> - DynamoDB v3.x is recommended for your local testing and development use.
> - Migration from DynamoDB local V2.x to V3.x requires updating import statements from `com.amazonaws.services.dynamodbv2` to `software.amazon.dynamodb` and updating Maven dependencies for Maven users.
> - If you're migrating an application that uses the SDK for Java v1.x to the SDK for Java 2.x, follow the steps for AWS SDK for Java 2.x.

Download DynamoDB local

Follow these steps to set up and run DynamoDB on your computer.

To set up DynamoDB on your computer

1. Download DynamoDB local for free from one of the following locations.

Download Links	Checksums
.tar.gz | .zip	.tar.gz.sha256 | .zip.sha256

> **Important**
>
> To run DynamoDB v2.6.0 or greater on your computer, you must have the Java Runtime Environment (JRE) version 17.x or newer. The application doesn't run on earlier JRE versions.

2. After you download the archive, extract the contents and copy the extracted directory to a location of your choice.
3. To start DynamoDB on your computer, open a command prompt window, navigate to the directory where you extracted `DynamoDBLocal.jar`, and enter the following command.

```
java -Djava.library.path=./DynamoDBLocal_lib -jar DynamoDBLocal.jar -sharedDb
```

> **Note**
>
> If you're using Windows PowerShell, be sure to enclose the parameter name or the entire name and value like this:
> `java -D"java.library.path=./DynamoDBLocal_lib" -jar DynamoDBLocal.jar`
> DynamoDB processes incoming requests until you stop it. To stop DynamoDB, press Ctrl+C at the command prompt.
> DynamoDB uses port 8000 by default. If port 8000 is unavailable, this command throws an exception. For a complete list of DynamoDB runtime options, including `-port`, enter this command.
> `java -Djava.library.path=./DynamoDBLocal_lib -jar DynamoDBLocal.jar -help`

4. Before you can access DynamoDB programmatically or through the AWS Command Line Interface (AWS CLI), you must configure your credentials to enable authorization for your applications. Downloadable DynamoDB requires any credentials to work, as shown in the following example.

   ```
   AWS Access Key ID: "fakeMyKeyId"
   AWS Secret Access Key: "fakeSecretAccessKey"
   Default Region Name: "fakeRegion"
   ```

   You can use the `aws configure` command of the AWS CLI to set up credentials. For more information, see Using the AWS CLI.

5. Start writing applications. To access DynamoDB running locally with the AWS CLI, use the `--endpoint-url` parameter. For example, use the following command to list DynamoDB tables.

   ```
   aws dynamodb list-tables --endpoint-url http://localhost:8000
   ```

Run DynamoDB local as Docker image

The downloadable version of Amazon DynamoDB is available as a Docker image. For more information, see dynamodb-local. To see your current DynamoDB local version, enter the following command:

```
java -Djava.library.path=./DynamoDBLocal_lib -jar DynamoDBLocal.jar -version
```

For an example of using DynamoDB local as part of a REST application built on the AWS Serverless Application Model (AWS SAM), see SAM DynamoDB application for managing orders. This sample application demonstrates how to use DynamoDB local for testing.

If you want to run a multi-container application that also uses the DynamoDB local container, use Docker Compose to define and run all the services in your application, including DynamoDB local.

To install and run DynamoDB local with Docker compose:

1. Download and install Docker desktop.
2. Copy the following code to a file and save it as `docker-compose.yml`.

   ```
   services:
    dynamodb-local:
      command: "-jar DynamoDBLocal.jar -sharedDb -dbPath ./data"
      image: "amazon/dynamodb-local:latest"
      container_name: dynamodb-local
      ports:
        - "8000:8000"
      volumes:
        - "./docker/dynamodb:/home/dynamodblocal/data"
      working_dir: /home/dynamodblocal
   ```

   If you want your application and DynamoDB local to be in separate containers, use the following yaml file.

   ```
   version: '3.8'
   services:
    dynamodb-local:
      command: "-jar DynamoDBLocal.jar -sharedDb -dbPath ./data"
      image: "amazon/dynamodb-local:latest"
      container_name: dynamodb-local
   ```

```
    ports:
      - "8000:8000"
    volumes:
      - "./docker/dynamodb:/home/dynamodblocal/data"
    working_dir: /home/dynamodblocal
  app-node:
    depends_on:
      - dynamodb-local
    image: amazon/aws-cli
    container_name: app-node
    ports:
     - "8080:8080"
    environment:
      AWS_ACCESS_KEY_ID: 'DUMMYIDEXAMPLE'
      AWS_SECRET_ACCESS_KEY: 'DUMMYEXAMPLEKEY'
    command:
      dynamodb describe-limits --endpoint-url http://dynamodb-local:8000 --region
  us-west-2
```

This docker-compose.yml script creates an `app-node` container and a `dynamodb-local` container. The script runs a command in the `app-node` container that uses the AWS CLI to connect to the `dynamodb-local` container and describes the account and table limits.

To use with your own application image, replace the `image` value in the example below with that of your application.

```
version: '3.8'
services:
  dynamodb-local:
    command: "-jar DynamoDBLocal.jar -sharedDb -dbPath ./data"
    image: "amazon/dynamodb-local:latest"
    container_name: dynamodb-local
    ports:
      - "8000:8000"
    volumes:
      - "./docker/dynamodb:/home/dynamodblocal/data"
    working_dir: /home/dynamodblocal
  app-node:
    image: location-of-your-dynamodb-demo-app:latest
    container_name: app-node
    ports:
      - "8080:8080"
```

```
  depends_on:
    - "dynamodb-local"
  links:
    - "dynamodb-local"
  environment:
    AWS_ACCESS_KEY_ID: 'DUMMYIDEXAMPLE'
    AWS_SECRET_ACCESS_KEY: 'DUMMYEXAMPLEKEY'
    REGION: 'eu-west-1'
```

> **Note**
>
> The YAML scripts require that you specify an AWS access key and an AWS secret key, but they are not required to be valid AWS keys for you to access DynamoDB local.

3. Run the following command-line command:

```
docker-compose up
```

Run DynamoDB local as an Apache Maven dependency

> **Note**
>
> If you're migrating an application that uses the SDK for Java v1.x to the SDK for Java 2.x, follow the steps for AWS SDK for Java 2.x.

Follow these steps to use Amazon DynamoDB in your application as a dependency.

To deploy DynamoDB as an Apache Maven repository

1. Download and install Apache Maven. For more information, see Downloading Apache Maven and Installing Apache Maven.
2. Add the DynamoDB Maven repository to your application's Project Object Model (POM) file.

```
<!--Dependency:-->
<dependencies>
   <dependency>
      <groupId>software.amazon.dynamodb</groupId>
      <artifactId>DynamoDBLocal</artifactId>
```

```
      <version>3.3.0</version>
   </dependency>
</dependencies>
```

Example template for use with Spring Boot 3 and/or Spring Framework 6:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://
maven.apache.org/xsd/maven-4.0.0.xsd">
<modelVersion>4.0.0</modelVersion>

<groupId>org.example</groupId>
<artifactId>SpringMavenDynamoDB</artifactId>
<version>1.0-SNAPSHOT</version>

<properties>
   <spring-boot.version>3.0.1</spring-boot.version>
   <maven.compiler.source>17</maven.compiler.source>
   <maven.compiler.target>17</maven.compiler.target>
   <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
</properties>

   <parent>
       <groupId>org.springframework.boot</groupId>
       <artifactId>spring-boot-starter-parent</artifactId>
       <version>3.1.0</version>
   </parent>

<dependencies>
   <dependency>
       <groupId>software.amazon.dynamodb</groupId>
       <artifactId>DynamoDBLocal</artifactId>
       <version>3.3.0</version>
   </dependency>
   <!-- Spring Boot -->
   <dependency>
       <groupId>org.springframework.boot</groupId>
       <artifactId>spring-boot-starter</artifactId>
       <version>${spring-boot.version}</version>
   </dependency>
   <!-- Spring Web -->
```

```
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
        <version>${spring-boot.version}</version>
    </dependency>
    <!-- Spring Data JPA -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactId>
        <version>${spring-boot.version}</version>
    </dependency>
    <!-- Other Spring dependencies -->
    <!-- Replace the version numbers with the desired version -->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>6.0.0</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-core</artifactId>
        <version>6.0.0</version>
    </dependency>
    <!-- Add other Spring dependencies as needed -->
    <!-- Add any other dependencies your project requires -->
</dependencies>
</project>
```

> **Note**
>
> You can also use the Maven central repository URL.

Run DynamoDB local in AWS CloudShell

AWS CloudShell is a browser-based, pre-authenticated shell that you can launch directly from the AWS Management Console. You can navigate to AWS CloudShell from the AWS Management Console a few different ways. For more information, see Getting started with AWS CloudShell.

Follow these steps to run DynamoDB local in your AWS CloudShell anywhere in the AWS Management Console.

To run DynamoDB local in your AWS CloudShell in the AWS Management Console

1. Launch AWS CloudShell from the console interface, choose an available AWS Region, and switch to your preferred shell, such as Bash, PowerShell, or Z shell.
2. To choose an AWS Region, go to the **Select a Region** menu and select a supported AWS Region. (Available Regions are highlighted.)
3. From the AWS Management Console, launch AWS CloudShell by choosing one of the following options:

   a. On the navigation bar, choose the **AWS CloudShell** icon.

   b. In the **Search** box, enter the word CloudShell, and then choose **CloudShell**.

   c. In the **Recently visited** widget, choose **CloudShell**.

   d. From the console toolbar, choose **CloudShell**.
4. To run DynamoDB local in AWS CloudShell you can use the `dynamodb-local` alias. You can specify additional command line options for changing DynamoDB local settings. See the section called "Usage notes" for available options.

   > **Note**
   >
   > To run DynamoDB local in the background, run DynamoDB local in AWS CloudShell using: `dynamodb-local &`.
5. To access DynamoDB running locally in AWS CloudShell with the AWS CLI, use the `--endpoint-url` parameter. For example, use the following command to list DynamoDB tables:

   ```
   aws dynamodb list-tables --endpoint-url http://localhost:8000
   ```

For an example of a sample project that showcases multiple approaches to set up and use DynamoDB local, including downloading JAR files, running it as a Docker image, and using it as a Maven dependency, see DynamoDB Local Sample Java Project.

DynamoDB local usage notes

Except for the endpoint, applications that run with the downloadable version of Amazon DynamoDB should also work with the DynamoDB web service. However, when using DynamoDB locally, you should be aware of the following:

- If you use the `-sharedDb` option, DynamoDB creates a single database file named *shared-local-instance.db*. Every program that connects to DynamoDB accesses this file. If you delete the file, you lose any data that you have stored in it.
- If you omit `-sharedDb`, the database file is named *myaccesskeyid_region.db*, with the AWS access key ID and AWS Region as they appear in your application configuration. If you delete the file, you lose any data that you have stored in it.
- If you use the `-inMemory` option, DynamoDB doesn't write any database files at all. Instead, all data is written to memory, and the data is not saved when you terminate DynamoDB.
- If you use the `-inMemory` option, the `-sharedDb` option is also required.
- If you use the `-optimizeDbBeforeStartup` option, you must also specify the `-dbPath` parameter so that DynamoDB can find its database file.
- The AWS SDKs for DynamoDB require that your application configuration specify an access key value and an AWS Region value. Unless you're using the `-sharedDb` or the `-inMemory` option, DynamoDB uses these values to name the local database file. These values don't have to be valid AWS values to run locally. However, you might find it convenient to use valid values so that you can run your code in the cloud later by changing the endpoint you're using.
- DynamoDB local always returns null for `billingModeSummary`.
- DynamoDB local `AWS_ACCESS_KEY_ID` can contain only letters (A–Z, a–z) and numbers (0–9).
- DynamoDB local doesn't support Point-in-time recovery (PITR).

Topics

- Command line options
- Setting the local endpoint
- Differences between downloadable DynamoDB and the DynamoDB web service

Command line options

You can use the following command line options with the downloadable version of DynamoDB:

- `-cors value` — Enables support for cross-origin resource sharing (CORS) for JavaScript. You must provide a comma-separated "allow" list of specific domains. The default setting for `-cors` is an asterisk (*), which allows public access.

- `-dbPath value` — The directory where DynamoDB writes its database file. If you don't specify this option, the file is written to the current directory. You can't specify both `-dbPath` and `-inMemory` at once.
- `-delayTransientStatuses` — Causes DynamoDB to introduce delays for certain operations. DynamoDB (downloadable version) can perform some tasks almost instantaneously, such as create/update/delete operations on tables and indexes. However, the DynamoDB service requires more time for these tasks. Setting this parameter helps DynamoDB running on your computer simulate the behavior of the DynamoDB web service more closely. (Currently, this parameter introduces delays only for global secondary indexes that are in either *CREATING* or *DELETING* status.)
- `-help` — Prints a usage summary and options.
- `-inMemory` — DynamoDB runs in memory instead of using a database file. When you stop DynamoDB, none of the data is saved. You can't specify both `-dbPath` and `-inMemory` at once.
- `-optimizeDbBeforeStartup` — Optimizes the underlying database tables before starting DynamoDB on your computer. You also must specify `-dbPath` when you use this parameter.
- `-port value` — The port number that DynamoDB uses to communicate with your application. If you don't specify this option, the default port is `8000`.

  > **Note**
  >
  > DynamoDB uses port 8000 by default. If port 8000 is unavailable, this command throws an exception. You can use the `-port` option to specify a different port number. For a complete list of DynamoDB runtime options, including `-port` , type this command: `java -Djava.library.path=./DynamoDBLocal_lib -jar DynamoDBLocal.jar -help`

- `-sharedDb` — If you specify `-sharedDb`, DynamoDB uses a single database file instead of separate files for each credential and Region.
- `-disableTelemetry` — When specified, DynamoDB local will not send any telemetry.
- `-version` — Prints the version of DynamoDB local.

Setting the local endpoint

By default, the AWS SDKs and tools use endpoints for the Amazon DynamoDB web service. To use the SDKs and tools with the downloadable version of DynamoDB, you must specify the local endpoint:

`http://localhost:8000`

AWS Command Line Interface

You can use the AWS Command Line Interface (AWS CLI) to interact with downloadable DynamoDB.

To access DynamoDB running locally, use the `--endpoint-url` parameter. The following is an example of using the AWS CLI to list the tables in DynamoDB on your computer.

```
aws dynamodb list-tables --endpoint-url http://localhost:8000
```

> **Note**
>
> The AWS CLI can't use the downloadable version of DynamoDB as a default endpoint. Therefore, you must specify `--endpoint-url` with each AWS CLI command.

AWS SDKs

The way you specify an endpoint depends on the programming language and AWS SDK you're using. The following sections describe how to do this:

- Java: Setting the AWS Region and endpoint (DynamoDB local supports the AWS SDK for Java V1 and V2)
- CodeSamples.Java.RegionAndEndpoint .NET: Setting the AWS Region and endpoint

Differences between downloadable DynamoDB and the DynamoDB web service

The downloadable version of DynamoDB is intended for development and testing purposes only. By comparison, the DynamoDB web service is a managed service with scalability, availability, and durability features that make it ideal for production use.

The downloadable version of DynamoDB differs from the web service in the following ways:

- AWS Regions and distinct AWS accounts are not supported at the client level.
- Provisioned throughput settings are ignored in downloadable DynamoDB, even though the `CreateTable` operation requires them. For `CreateTable`, you can specify any numbers you want for provisioned read and write throughput, even though these numbers are not used. You can call `UpdateTable` as many times as you want per day. However, any changes to provisioned throughput values are ignored.
- `Scan` operations are performed sequentially. Parallel scans are not supported. The `Segment` and `TotalSegments` parameters of the `Scan` operation are ignored.
- The speed of read and write operations on table data is limited only by the speed of your computer. `CreateTable`, `UpdateTable`, and `DeleteTable` operations occur immediately, and table state is always ACTIVE. `UpdateTable` operations that change only the provisioned throughput settings on tables or global secondary indexes occur immediately. If an `UpdateTable` operation creates or deletes any global secondary indexes, then those indexes transition through normal states (such as CREATING and DELETING, respectively) before they become an ACTIVE state. The table remains ACTIVE during this time.
- Read operations are eventually consistent. However, due to the speed of DynamoDB local running on your computer, most reads appear to be strongly consistent.
- Item collection metrics and item collection sizes are not tracked. In operation responses, nulls are returned instead of item collection metrics.
- In DynamoDB, there is a 1 MB limit on data returned per result set. Both the DynamoDB web service and the downloadable version enforce this limit. However, when querying an index, the DynamoDB service calculates only the size of the projected key and attributes. By contrast, the downloadable version of DynamoDB calculates the size of the entire item.
- If you're using DynamoDB Streams, the rate at which shards are created might differ. In the DynamoDB web service, shard-creation behavior is partially influenced by table partition activity. When you run DynamoDB locally, there is no table partitioning. In either case, shards are ephemeral, so your application should not be dependent on shard behavior.
- `TransactionConflictExceptions` aren't thrown by downloadable DynamoDB for transactional APIs. We recommend that you use a Java mocking framework to simulate `TransactionConflictExceptions` in the DynamoDB handler to test how your application responds to conflicting transactions.
- In the DynamoDB web service, whether being accessed via the console or the AWS CLI, table names are case sensitive. A table named `Authors` and one named `authors` can both exist as

separate tables. In the downloadable version, table names are case insensitive, and attempting to create these two tables would result in an error.

- Tagging is not supported in the downloadable version of DynamoDB.
- The downloadable version of DynamoDB ignores the Limit parameter in ExecuteStatement.

Release history for DynamoDB local

The following table describes the important changes in each release of *DynamoDB local*.

Version	Change	Description	Date
3.3.0	Adding multi-attribute key support for Global Secondary Indexes	• Adding multi-attribute key support for Global Secondary Indexes • Updating SDK Java version to the latest public version from 2.41.0 to 2.41.7	January 19, 2026
3.2.0	Fixed Compatibility Issues with multiple Kotlin Versions	• Adding Support for ShardFilter param on DescribeStream API • Updating SDK Java version to the latest public version from 2.33.0 to 2.41.0 • Fixing DeletionProtection bug on UpdateTable	January 09, 2026
3.1.0	Improving performance for PartiQL	• Updating SDK Java version to	September 14, 2025

Version	Change	Description	Date
	Queries, Including Joda-time dependency	the latest public version from 2.25.50 to 2.33.0 • Including Joda-time dependency inside Pom.xml file • Improving performance for PartiQL queries • Upgrading dependencies to fix multiple CVE vulnerability issues	
3.0.0	Migrating from AWS SDK Java V1 to V2	• Migrating from AWS SDK Java V1 to V2 • Updated package structure from `com.amazonaws.services.dynamodbv2` to `software.amazon.dynamodb.services` • Removed AWS SDK Java V1 dependencies	July 17, 2025

Version	Change	Description	Date
2.6.0	Support table ARN as table name in DynamoDB APIs Performance fix and security updates	• Added support for using table ARN as table name in several DynamoDB APIs • Fixing `CreateStreamTable` bug on high performance machines, such as Mac M3 • Upgrading dependencies to fix vulnerability issues (CVE-2022-49043,CVE-2024-56732, CVE-2020-29582, CVE-2025-21502, CVE-2024-50602, CVE-2025-24970, CVE-2025-25193)	March 13, 2025
2.5.4	Upgrading to Jetty Dependencies	• Upgrading from Jetty 12.0.8 to Jetty 12.0.14 (Resolves CVE-2024-6763, CVE-2024-8184, CVE-2024-47535) Mitigation for (CVE-2024-21634)	December 12, 2024

Version	Change	Description	Date
2.5.3	Upgrading Jackson Dependencies to 2.17.x in Log4j Core (Resolves CVE-2022-1471)	• Upgrading Jackson Dependencies to 2.17.x in Log4j Core (Resolves CVE-2022-1471) to address a critical security vulnerability in the SnakeYAML library, which is a transitive dependency	November 6, 2024
2.5.2	Bug fix for Update table workflow	• Bug fix for workflow when update table tries to update table with Billing Mode Ondemand to Provisioned With GSI	June 20, 2024
2.5.1	Patch for bugs introduced in `OndemandThroughPut` feature	• Fixed a couple of bugs related to `OndemandThroughPut`	June 5, 2024

Version	Change	Description	Date
2.5.0	Support for configurable maximum throughput for on-demand tables, `ReturnValuesOnConditionCheckFailure`, `BatchExecuteStatement`, and `ExecuteTransactionRequest`	• Adding telemetry to Embedded Mode • Fixing the SDKv2 translation for `ConditionalCheckException`	May 28, 2024
2.4.0	Support for `ReturnValuesOnConditionCheckFailure` - Embedded Mode	• Embedded Mode Fix for `TrimmedDataAccessException` for Operation on Multiple Streams • Fixing exception translation for SDKv2 in Embedded Mode	April 17, 2024
2.3.0	Jetty and JDK Upgrade	• Upgrading to Jetty 12.0.2 • Upgrading to JDK 17 • Upgrading ANTLR4 to 4.10.1	March 14, 2024

Version	Change	Description	Date
2.2.0	Added support for table deletion protection and the `ReturnValuesOnConditionCheckFailure` parameter	• Added support of Table delete protection • Added support for ReturnValuesOnConditionCheckFailure • Added support for -version flag	December 14, 2023
2.1.0	Support for SQLLite Native Libraries for Maven projects and adding telemetry	• Adding telemetry to DynamoDB local • Dynamically copy SQLLite Native Libraries for Maven projects • Removed io.github.ganadist.sqlite4java library from Maven dependency • Upgrading GoogleGuava to 32.1.1-jre	October 23, 2023

Version	Change	Description	Date
2.0.0	Migrating from javax to jakarta namespace and JDK11 Support	• Migrating from javax to jakarta namespace and JDK11 support • Fix for handling invalid access and secret key while server startup • Fixing Maven identified vulnerabilities by updating dependencies	July 5, 2023
1.25.1	Upgrading Jackson Dependencies to 2.17.x in Log4j Core (Resolves CVE-2022-1471)	Upgrading Jackson Dependencies to 2.17.x in Log4j Core (Resolves CVE-2022-1471) to address a critical security vulnerability in the SnakeYAML library, which is a transitive dependency	November 6, 2024
1.25.0	Added support for table deletion protection and the `ReturnValuesOnConditionCheckFailure` parameter	• Added support of Table delete protection • Added support for ReturnValuesOnConditionCheckFailure • Added support for -version flag	December 18, 2023

Version	Change	Description	Date
1.24.0	Support for SQLLite Native Libraries for Maven projects and adding telemetry	• Adding telemetry to DynamoDB local • Dynamically copy SQLLite Native Libraries for Maven projects • Removed io.github.ganadist.sqlite4java library from Maven dependency • Upgrading GoogleGuava to 32.1.1-jre	October 23, 2023
1.23.0	Handle invalid access and secret key while server startup	• Fix for handling invalid access and secret key while server startup • Fixing Maven identified vulnerabilities by updating dependencies	June 28, 2023
1.22.0	Support of Limit Operation for PartiQL	• Optimize IN clause for PartiQL • Support for Limit Operation • M1 support for Maven projects	June 8, 2023

Version	Change	Description	Date
1.21.0	Support for 100 actions per transacti on	• Increased actions per transaction from 25 to 100 • Upgrading docker image Open JDK to 11 • Fixing the parity for exception thrown when duplicate items in BatchExec uteStatement	January 26, 2023
1.20.0	Added support for M1 Mac	• Added support for M1 Mac • Upgrading Jetty dependency to 9.4.48.v20220622	September 12, 2022
1.19.0	Upgraded the PartiQL Parser	Upgraded the PartiQL Parser and other related libraries	July 27, 2022
1.18.0	Upgraded log4j-core and Jackson-core	Upgraded log4j-core to 2.17.1 and Jackson-core 2.10.x to 2.12.0	January 10, 2022
1.17.2	Upgraded log4j-core	Upgraded log4j-cor e dependency to version 2.16	January 16, 2021

Version	Change	Description	Date
1.17.1	Upgraded log4j-core	Updated log4j-core dependency to patch zero-day exploit to prevent remote code execution - Log4Shel	January 10, 2021
1.17.0	Deprecated Javascript Web Shell	• Updated the AWS SDK dependency to AWS SDK for Java 1.12.x • Deprecated Javascript Web Shell	January 8, 2021

Telemetry in DynamoDB local

At AWS, we develop and launch services based on what we learn from interactions with customers, and we use customer feedback to iterate on our products. Telemetry is additional information that helps us to better understand our customers needs, diagnose issues, and deliver features that improve the customer experience.

DynamoDB local collects telemetry, such as generic usage metrics, systems and environment information, and errors. For details about the types of telemetry collected, see Types of information collected.

DynamoDB local does not collect personal information, such as user names or email addresses. It also does not extract sensitive project-level information.

As a customer, you control whether telemetry is turned on, and you can change your settings at any point in time. If telemetry remains on, DynamoDB local sends telemetry data in the background without requiring any additional customer interaction.

Turn off telemetry using command line options

You can turn off telemetry using command line options when starting DynamoDB local using the option `-disableTelemetry`. For more information, see Command line options.

Turn off telemetry for a single session

In macOS and Linux operating systems, you can turn off telemetry for a single session. To turn off telemetry for your current session, run the following command to set the environment variable `DDB_LOCAL_TELEMETRY` to `false`. Repeat the command for each new terminal or session.

```
export DDB_LOCAL_TELEMETRY=0
```

Turn off telemetry for your profile in all sessions

Run the following commands to turn off telemetry for all sessions when you're running DynamoDB local on your operating system.

To turn off telemetry in Linux

1. Run:

   ```
   echo "export DDB_LOCAL_TELEMETRY=0" >>~/.profile
   ```

2. Run:

   ```
   source ~/.profile
   ```

To turn off telemetry in macOS

1. Run:

   ```
   echo "export DDB_LOCAL_TELEMETRY=0" >>~/.profile
   ```

2. Run:

   ```
   source ~/.profile
   ```

To turn off telemetry in Windows

1. Run:

   ```
   setx DDB_LOCAL_TELEMETRY 0
   ```

2. Run:

```
refreshenv
```

Turn off telemetry using DynamoDB local embedded on Maven projects

You can turn off telemetry using DynamoDB local embedded on Maven projects.

```
boolean disableTelemetry = true;
// AWS SDK v1
 AmazonDynamoDB amazonDynamoDB =
 DynamoDBEmbedded.create(disableTelemetry).amazonDynamoDB();

// AWS SDK v2
DynamoDbClient ddbClientSDKv2Local =
 DynamoDBEmbedded.create(disableTelemetry).dynamoDbClient();
```

Types of information collected

- **Usage information** — The generic telemetry like server start/stop and the API or Operation called.
- **System and environment information** — The Java version, operating system (Windows, Linux or macOS), the environment in which DynamoDB local runs (for example, Stand alone JAR, Docker container, or as a Maven Dependency), and hash values of usage attributes.

Learn more

The telemetry data that DynamoDB local collects adheres to the AWS data privacy policies. For more information, see the following:

- AWS service terms
- Data privacy FAQ

Step 1: Create a table in DynamoDB

In this step, you create a `Music` table in Amazon DynamoDB. The table has the following details:

- Partition key — `Artist`

- Sort key — `SongTitle`

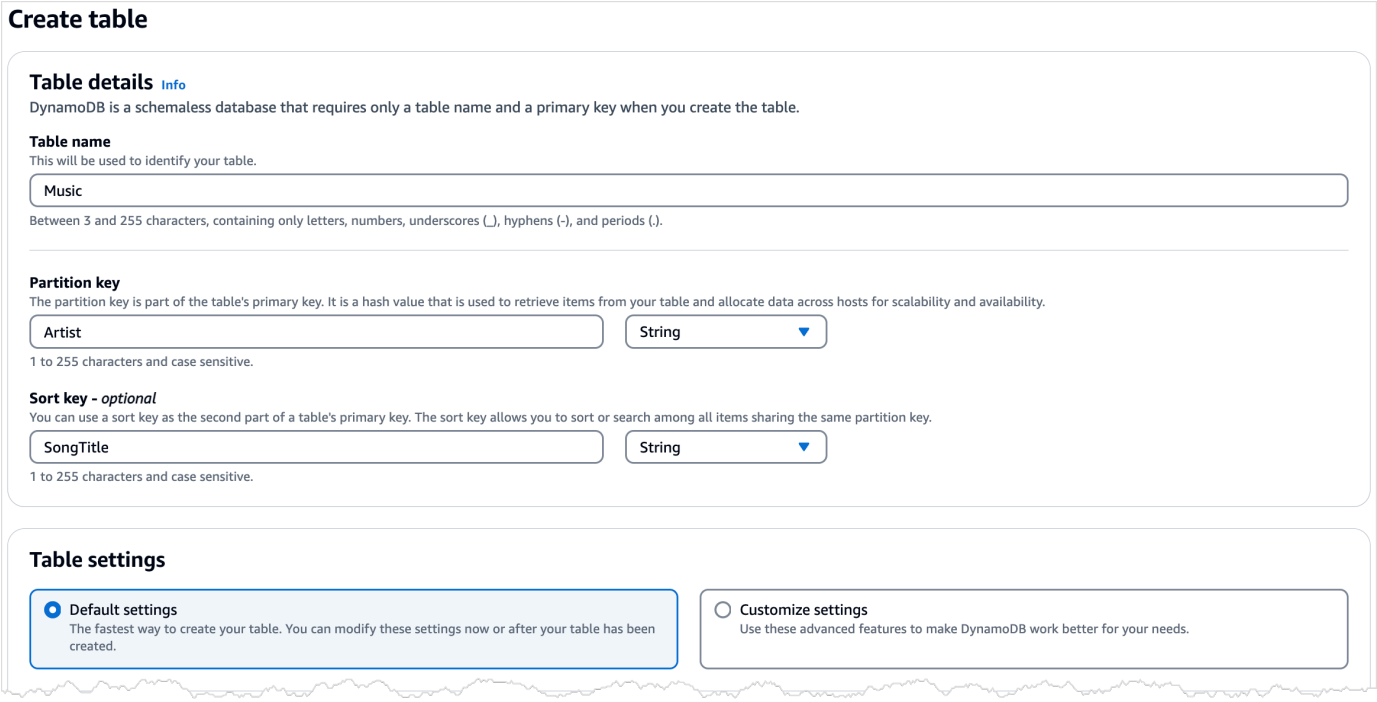
For more information about table operations, see Working with tables and data in DynamoDB.

> **Note**
>
> Before you begin, make sure that you followed the steps in Prerequisites.

AWS Management Console

To create a new `Music` table using the DynamoDB console:

1. Sign in to the AWS Management Console and open the DynamoDB console at https://console.aws.amazon.com/dynamodb/.
2. In the left navigation pane, choose **Tables**.
3. Choose **Create table**.
4. Enter the **Table details** as follows:
   a. For **Table name**, enter **`Music`**.
   b. For **Partition key**, enter **`Artist`**.
   c. For **Sort key**, enter **`SongTitle`**.
5. For **Table settings**, keep the default selection of **Default settings**.
6. Choose **Create table** to create the table.



7. Once the table is in `ACTIVE` status, we recommend that you enable Point-in-time backups for DynamoDB on the table by performing the following steps:

   a. Choose the table name to open the table.

   b. Choose **Backups**.

   c. Choose **Edit** in the **Point-in-time recovery (PITR)** section.

   d. On the **Edit point-in-time recovery settings** page, choose **Turn on point-in-time recovery**.

   e. Choose **Save changes**.

AWS CLI

The following AWS CLI example creates a new `Music` table using `create-table`.

Linux

```
aws dynamodb create-table \
    --table-name Music \
    --attribute-definitions \
        AttributeName=Artist,AttributeType=S \
        AttributeName=SongTitle,AttributeType=S \
```

```
    --key-schema AttributeName=Artist,KeyType=HASH
 AttributeName=SongTitle,KeyType=RANGE \
    --billing-mode PAY_PER_REQUEST \
    --table-class STANDARD
```

Windows CMD

```
aws dynamodb create-table ^
    --table-name Music ^
    --attribute-definitions ^
        AttributeName=Artist,AttributeType=S ^
        AttributeName=SongTitle,AttributeType=S ^
    --key-schema AttributeName=Artist,KeyType=HASH
 AttributeName=SongTitle,KeyType=RANGE ^
    --billing-mode PAY_PER_REQUEST ^
    --table-class STANDARD
```

Using `create-table` returns the following sample result.

```
{
    "TableDescription": {
        "AttributeDefinitions": [
            {
                "AttributeName": "Artist",
                "AttributeType": "S"
            },
            {
                "AttributeName": "SongTitle",
                "AttributeType": "S"
            }
        ],
        "TableName": "Music",
        "KeySchema": [
            {
                "AttributeName": "Artist",
                "KeyType": "HASH"
            },
            {
                "AttributeName": "SongTitle",
                "KeyType": "RANGE"
            }
        ],
        "TableStatus": "CREATING",
```

```
        "CreationDateTime": "2023-03-29T12:11:43.379000-04:00",
        "ProvisionedThroughput": {
            "NumberOfDecreasesToday": 0,
            "ReadCapacityUnits": 5,
            "WriteCapacityUnits": 5
        },
        "TableSizeBytes": 0,
        "ItemCount": 0,
        "TableArn": "arn:aws:dynamodb:us-east-1:111122223333:table/Music",
        "TableId": "60abf404-1839-4917-a89b-a8b0ab2a1b87",
        "TableClassSummary": {
            "TableClass": "STANDARD"
        }
    }
}
}
```

Note that the value of the `TableStatus` field is set to `CREATING`.

To verify that DynamoDB has finished creating the `Music` table, use the `describe-table` command.

Linux

```
aws dynamodb describe-table --table-name Music | grep TableStatus
```

Windows CMD

```
aws dynamodb describe-table --table-name Music | findstr TableStatus
```

This command returns the following result. When DynamoDB finishes creating the table, the value of the `TableStatus` field is set to `ACTIVE`.

```
"TableStatus": "ACTIVE",
```

Once the table is in `ACTIVE` status, it's considered best practice to enable Point-in-time backups for DynamoDB on the table by running the following command:

Linux

```
aws dynamodb update-continuous-backups \
    --table-name Music \
    --point-in-time-recovery-specification \
        PointInTimeRecoveryEnabled=true
```

Windows CMD

```
aws dynamodb update-continuous-backups --table-name Music --point-in-time-recovery-
specification PointInTimeRecoveryEnabled=true
```

This command returns the following result.

```
{
    "ContinuousBackupsDescription": {
        "ContinuousBackupsStatus": "ENABLED",
        "PointInTimeRecoveryDescription": {
            "PointInTimeRecoveryStatus": "ENABLED",
            "EarliestRestorableDateTime": "2023-03-29T12:18:19-04:00",
            "LatestRestorableDateTime": "2023-03-29T12:18:19-04:00"
        }
    }
}
```

Note

There are cost implications to enabling continuous backups with point-in-time recovery. For more information about pricing, see Amazon DynamoDB pricing.

AWS SDK

The following code examples show how to create a DynamoDB table using an AWS SDK.

.NET

SDK for .NET (v4)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    /// <summary>
    /// Creates a new Amazon DynamoDB table and then waits for the new
    /// table to become active.
    /// </summary>
    /// <param name="tableName">The name of the table to create.</param>
    /// <returns>A Boolean value indicating the success of the operation.</
returns>
    public async Task<bool> CreateMovieTableAsync(string tableName)
    {
        try
        {
            var response = await _amazonDynamoDB.CreateTableAsync(new
 CreateTableRequest
            {
                TableName = tableName,
                AttributeDefinitions = new List<AttributeDefinition>()
                {
                    new AttributeDefinition
                    {
                        AttributeName = "title",
                        AttributeType = ScalarAttributeType.S,
                    },
                    new AttributeDefinition
                    {
                        AttributeName = "year",
                        AttributeType = ScalarAttributeType.N,
                    },
                },
                KeySchema = new List<KeySchemaElement>()
                {
                    new KeySchemaElement
```

```
                    {
                        AttributeName = "year",
                        KeyType = KeyType.HASH,
                    },
                    new KeySchemaElement
                    {
                        AttributeName = "title",
                        KeyType = KeyType.RANGE,
                    },
                },
                BillingMode = BillingMode.PAY_PER_REQUEST,
            });

            // Wait until the table is ACTIVE and then report success.
            Console.Write("Waiting for table to become active...");

            var request = new DescribeTableRequest
            {
                TableName = response.TableDescription.TableName,
            };

            TableStatus status;

            int sleepDuration = 2000;

            do
            {
                Thread.Sleep(sleepDuration);

                var describeTableResponse = await
_amazonDynamoDB.DescribeTableAsync(request);
                status = describeTableResponse.Table.TableStatus;

                Console.Write(".");
            }
            while (status != "ACTIVE");

            return status == TableStatus.ACTIVE;
        }
        catch (ResourceInUseException ex)
        {
            Console.WriteLine($"Table {tableName} already exists. {ex.Message}");
            throw;
        }
```

```
        catch (AmazonDynamoDBException ex)
        {
            Console.WriteLine($"An Amazon DynamoDB error occurred while creating
table {tableName}. {ex.Message}");
            throw;
        }
        catch (Exception ex)
        {
            Console.WriteLine($"An error occurred while creating table
{tableName}. {ex.Message}");
            throw;
        }
    }
```

- For API details, see CreateTable in *AWS SDK for .NET API Reference*.

Bash

AWS CLI with Bash script

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
###############################################################################
# function dynamodb_create_table
#
# This function creates an Amazon DynamoDB table.
#
# Parameters:
#       -n table_name  -- The name of the table to create.
#       -a attribute_definitions -- JSON file path of a list of attributes and
 their types.
#       -k key_schema -- JSON file path of a list of attributes and their key
 types.
#
#  Returns:
#       0 - If successful.
```

```
#       1 - If it fails.
###############################################################################
function dynamodb_create_table() {
  local table_name attribute_definitions key_schema response
  local option OPTARG # Required to use getopts command in a function.

  ######################################
  # Function usage explanation
  ######################################
  function usage() {
    echo "function dynamodb_create_table"
    echo "Creates an Amazon DynamoDB table with on-demand billing."
    echo " -n table_name  -- The name of the table to create."
    echo " -a attribute_definitions -- JSON file path of a list of attributes and
 their types."
    echo " -k key_schema -- JSON file path of a list of attributes and their key
 types."
    echo ""
  }

  # Retrieve the calling parameters.
  while getopts "n:a:k:h" option; do
    case "${option}" in
      n) table_name="${OPTARG}" ;;
      a) attribute_definitions="${OPTARG}" ;;
      k) key_schema="${OPTARG}" ;;
      h)
        usage
        return 0
        ;;
      \?)
        echo "Invalid parameter"
        usage
        return 1
        ;;
    esac
  done
  export OPTIND=1

  if [[ -z "$table_name" ]]; then
    errecho "ERROR: You must provide a table name with the -n parameter."
    usage
    return 1
  fi
```

```
  if [[ -z "$attribute_definitions" ]]; then
    errecho "ERROR: You must provide an attribute definitions json file path the
 -a parameter."
    usage
    return 1
  fi

  if [[ -z "$key_schema" ]]; then
    errecho "ERROR: You must provide a key schema json file path the -k
 parameter."
    usage
    return 1
  fi

  iecho "Parameters:\n"
  iecho "    table_name:   $table_name"
  iecho "    attribute_definitions:   $attribute_definitions"
  iecho "    key_schema:   $key_schema"
  iecho ""

  response=$(aws dynamodb create-table \
    --table-name "$table_name" \
    --attribute-definitions file://"$attribute_definitions" \
    --billing-mode PAY_PER_REQUEST \
    --key-schema file://"$key_schema" )

  local error_code=${?}

  if [[ $error_code -ne 0 ]]; then
    aws_cli_error_log $error_code
    errecho "ERROR: AWS reports create-table operation failed.$response"
    return 1
  fi

  return 0
}
```

The utility functions used in this example.

```
###############################################################################
# function iecho
```

```
#
# This function enables the script to display the specified text only if
# the global variable $VERBOSE is set to true.
##############################################################################
function iecho() {
  if [[ $VERBOSE == true ]]; then
    echo "$@"
  fi
}

##############################################################################
# function errecho
#
# This function outputs everything sent to it to STDERR (standard error output).
##############################################################################
function errecho() {
  printf "%s\n" "$*" 1>&2
}

##############################################################################
# function aws_cli_error_log()
#
# This function is used to log the error messages from the AWS CLI.
#
# See https://docs.aws.amazon.com/cli/latest/topic/return-codes.html#cli-aws-
help-return-codes.
#
# The function expects the following argument:
#         $1 - The error code returned by the AWS CLI.
#
#  Returns:
#          0: - Success.
#
##############################################################################
function aws_cli_error_log() {
  local err_code=$1
  errecho "Error code : $err_code"
  if [ "$err_code" == 1 ]; then
    errecho "  One or more S3 transfers failed."
  elif [ "$err_code" == 2 ]; then
    errecho "  Command line failed to parse."
  elif [ "$err_code" == 130 ]; then
    errecho "  Process received SIGINT."
  elif [ "$err_code" == 252 ]; then
```

```
    errecho "  Command syntax invalid."
  elif [ "$err_code" == 253 ]; then
    errecho "  The system environment or configuration was invalid."
  elif [ "$err_code" == 254 ]; then
    errecho "  The service returned an error."
  elif [ "$err_code" == 255 ]; then
    errecho "  255 is a catch-all error."
  fi

  return 0
}
```

- For API details, see CreateTable in *AWS CLI Command Reference*.

C++

SDK for C++

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
//! Create an Amazon DynamoDB table.
/*!
  \sa createTable()
  \param tableName: Name for the DynamoDB table.
  \param primaryKey: Primary key for the DynamoDB table.
  \param clientConfiguration: AWS client configuration.
  \return bool: Function succeeded.
 */
bool AwsDoc::DynamoDB::createTable(const Aws::String &tableName,
                                   const Aws::String &primaryKey,
                                   const Aws::Client::ClientConfiguration
 &clientConfiguration) {
    Aws::DynamoDB::DynamoDBClient dynamoClient(clientConfiguration);

    std::cout << "Creating table " << tableName <<
              " with a simple primary key: \"" << primaryKey << "\"." <<
 std::endl;
```

```
    Aws::DynamoDB::Model::CreateTableRequest request;

    Aws::DynamoDB::Model::AttributeDefinition hashKey;
    hashKey.SetAttributeName(primaryKey);
    hashKey.SetAttributeType(Aws::DynamoDB::Model::ScalarAttributeType::S);
    request.AddAttributeDefinitions(hashKey);

    Aws::DynamoDB::Model::KeySchemaElement keySchemaElement;
    keySchemaElement.WithAttributeName(primaryKey).WithKeyType(
            Aws::DynamoDB::Model::KeyType::HASH);
    request.AddKeySchema(keySchemaElement);

    Aws::DynamoDB::Model::ProvisionedThroughput throughput;
    throughput.WithReadCapacityUnits(5).WithWriteCapacityUnits(5);
    request.SetProvisionedThroughput(throughput);
    request.SetTableName(tableName);

    const Aws::DynamoDB::Model::CreateTableOutcome &outcome =
 dynamoClient.CreateTable(
            request);
    if (outcome.IsSuccess()) {
        std::cout << "Table \""
                  << outcome.GetResult().GetTableDescription().GetTableName() <<
                  " created!" << std::endl;
    }
    else {
        std::cerr << "Failed to create table: " <<
 outcome.GetError().GetMessage()
                  << std::endl;
        return false;
    }

    return waitTableActive(tableName, dynamoClient);
}
```

Code that waits for the table to become active.

```
//! Query a newly created DynamoDB table until it is active.
/*!
  \sa waitTableActive()
  \param waitTableActive: The DynamoDB table's name.
```

```
  \param dynamoClient: A DynamoDB client.
  \return bool: Function succeeded.
*/
bool AwsDoc::DynamoDB::waitTableActive(const Aws::String &tableName,
                                       const Aws::DynamoDB::DynamoDBClient
 &dynamoClient) {

    // Repeatedly call DescribeTable until table is ACTIVE.
    const int MAX_QUERIES = 20;
    Aws::DynamoDB::Model::DescribeTableRequest request;
    request.SetTableName(tableName);

    int count = 0;
    while (count < MAX_QUERIES) {
        const Aws::DynamoDB::Model::DescribeTableOutcome &result =
 dynamoClient.DescribeTable(
                request);
        if (result.IsSuccess()) {
            Aws::DynamoDB::Model::TableStatus status =
 result.GetResult().GetTable().GetTableStatus();

            if (Aws::DynamoDB::Model::TableStatus::ACTIVE != status) {
                std::this_thread::sleep_for(std::chrono::seconds(1));
            }
            else {
                return true;
            }
        }
        else {
            std::cerr << "Error DynamoDB::waitTableActive "
                      << result.GetError().GetMessage() << std::endl;
            return false;
        }
        count++;
    }
    return false;
}
```

- For API details, see CreateTable in *AWS SDK for C++ API Reference*.

CLI

AWS CLI

Example 1: To create a table with tags

The following `create-table` example uses the specified attributes and key schema to create a table named `MusicCollection`. This table uses provisioned throughput and is encrypted at rest using the default AWS owned CMK. The command also applies a tag to the table, with a key of `Owner` and a value of `blueTeam`.

```
aws dynamodb create-table \
    --table-name MusicCollection \
    --attribute-
definitions AttributeName=Artist,AttributeType=S AttributeName=SongTitle,AttributeType=S
 \
    --key-
schema AttributeName=Artist,KeyType=HASH AttributeName=SongTitle,KeyType=RANGE \
    --provisioned-throughput ReadCapacityUnits=5,WriteCapacityUnits=5 \
    --tags Key=Owner,Value=blueTeam
```

Output:

```
{
    "TableDescription": {
        "AttributeDefinitions": [
            {
                "AttributeName": "Artist",
                "AttributeType": "S"
            },
            {
                "AttributeName": "SongTitle",
                "AttributeType": "S"
            }
        ],
        "ProvisionedThroughput": {
            "NumberOfDecreasesToday": 0,
            "WriteCapacityUnits": 5,
            "ReadCapacityUnits": 5
        },
        "TableSizeBytes": 0,
        "TableName": "MusicCollection",
        "TableStatus": "CREATING",
```

```
        "KeySchema": [
            {
                "KeyType": "HASH",
                "AttributeName": "Artist"
            },
            {
                "KeyType": "RANGE",
                "AttributeName": "SongTitle"
            }
        ],
        "ItemCount": 0,
        "CreationDateTime": "2020-05-26T16:04:41.627000-07:00",
        "TableArn": "arn:aws:dynamodb:us-west-2:123456789012:table/
MusicCollection",
        "TableId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111"
    }
}
```

For more information, see Basic Operations for Tables in the *Amazon DynamoDB Developer Guide*.

Example 2: To create a table in On-Demand Mode

The following example creates a table called `MusicCollection` using on-demand mode, rather than provisioned throughput mode. This is useful for tables with unpredictable workloads.

```
aws dynamodb create-table \
    --table-name MusicCollection \
    --attribute-
definitions AttributeName=Artist,AttributeType=S AttributeName=SongTitle,AttributeType=S
 \
    --key-
schema AttributeName=Artist,KeyType=HASH AttributeName=SongTitle,KeyType=RANGE \
    --billing-mode PAY_PER_REQUEST
```

Output:

```
{
    "TableDescription": {
        "AttributeDefinitions": [
            {
                "AttributeName": "Artist",
```

```
                "AttributeType": "S"
            },
            {
                "AttributeName": "SongTitle",
                "AttributeType": "S"
            }
        ],
        "TableName": "MusicCollection",
        "KeySchema": [
            {
                "AttributeName": "Artist",
                "KeyType": "HASH"
            },
            {
                "AttributeName": "SongTitle",
                "KeyType": "RANGE"
            }
        ],
        "TableStatus": "CREATING",
        "CreationDateTime": "2020-05-27T11:44:10.807000-07:00",
        "ProvisionedThroughput": {
            "NumberOfDecreasesToday": 0,
            "ReadCapacityUnits": 0,
            "WriteCapacityUnits": 0
        },
        "TableSizeBytes": 0,
        "ItemCount": 0,
        "TableArn": "arn:aws:dynamodb:us-west-2:123456789012:table/
MusicCollection",
        "TableId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111",
        "BillingModeSummary": {
            "BillingMode": "PAY_PER_REQUEST"
        }
    }
}
```

For more information, see Basic Operations for Tables in the *Amazon DynamoDB Developer Guide*.

Example 3: To create a table and encrypt it with a Customer Managed CMK

The following example creates a table named `MusicCollection` and encrypts it using a customer managed CMK.

```
aws dynamodb create-table \
    --table-name MusicCollection \
    --attribute-
definitions AttributeName=Artist,AttributeType=S AttributeName=SongTitle,AttributeType=S
 \
    --key-
schema AttributeName=Artist,KeyType=HASH AttributeName=SongTitle,KeyType=RANGE \
    --provisioned-throughput ReadCapacityUnits=5,WriteCapacityUnits=5 \
    --sse-specification Enabled=true,SSEType=KMS,KMSMasterKeyId=abcd1234-
abcd-1234-a123-ab1234a1b234
```

Output:

```
{
    "TableDescription": {
        "AttributeDefinitions": [
            {
                "AttributeName": "Artist",
                "AttributeType": "S"
            },
            {
                "AttributeName": "SongTitle",
                "AttributeType": "S"
            }
        ],
        "TableName": "MusicCollection",
        "KeySchema": [
            {
                "AttributeName": "Artist",
                "KeyType": "HASH"
            },
            {
                "AttributeName": "SongTitle",
                "KeyType": "RANGE"
            }
        ],
        "TableStatus": "CREATING",
        "CreationDateTime": "2020-05-27T11:12:16.431000-07:00",
        "ProvisionedThroughput": {
            "NumberOfDecreasesToday": 0,
            "ReadCapacityUnits": 5,
            "WriteCapacityUnits": 5
        },
```

```
        "TableSizeBytes": 0,
        "ItemCount": 0,
        "TableArn": "arn:aws:dynamodb:us-west-2:123456789012:table/
MusicCollection",
        "TableId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111",
        "SSEDescription": {
            "Status": "ENABLED",
            "SSEType": "KMS",
            "KMSMasterKeyArn": "arn:aws:kms:us-west-2:123456789012:key/abcd1234-
abcd-1234-a123-ab1234a1b234"
        }
    }
}
```

For more information, see Basic Operations for Tables in the *Amazon DynamoDB Developer Guide*.

Example 4: To create a table with a Local Secondary Index

The following example uses the specified attributes and key schema to create a table named `MusicCollection` with a Local Secondary Index named `AlbumTitleIndex`.

```
aws dynamodb create-table \
    --table-name MusicCollection \
    --attribute-
definitions AttributeName=Artist,AttributeType=S AttributeName=SongTitle,AttributeType=S
 \
    --key-
schema AttributeName=Artist,KeyType=HASH AttributeName=SongTitle,KeyType=RANGE \
    --provisioned-throughput ReadCapacityUnits=10,WriteCapacityUnits=5 \
    --local-secondary-indexes \
        "[
            {
                \"IndexName\": \"AlbumTitleIndex\",
                \"KeySchema\": [
                    {\"AttributeName\": \"Artist\",\"KeyType\":\"HASH\"},
                    {\"AttributeName\": \"AlbumTitle\",\"KeyType\":\"RANGE\"}
                ],
                \"Projection\": {
                    \"ProjectionType\": \"INCLUDE\",
                    \"NonKeyAttributes\": [\"Genre\", \"Year\"]
                }
            }
```

```
        ]"
```

Output:

```
{
    "TableDescription": {
        "AttributeDefinitions": [
            {
                "AttributeName": "AlbumTitle",
                "AttributeType": "S"
            },
            {
                "AttributeName": "Artist",
                "AttributeType": "S"
            },
            {
                "AttributeName": "SongTitle",
                "AttributeType": "S"
            }
        ],
        "TableName": "MusicCollection",
        "KeySchema": [
            {
                "AttributeName": "Artist",
                "KeyType": "HASH"
            },
            {
                "AttributeName": "SongTitle",
                "KeyType": "RANGE"
            }
        ],
        "TableStatus": "CREATING",
        "CreationDateTime": "2020-05-26T15:59:49.473000-07:00",
        "ProvisionedThroughput": {
            "NumberOfDecreasesToday": 0,
            "ReadCapacityUnits": 10,
            "WriteCapacityUnits": 5
        },
        "TableSizeBytes": 0,
        "ItemCount": 0,
        "TableArn": "arn:aws:dynamodb:us-west-2:123456789012:table/
MusicCollection",
        "TableId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111",
```

```
        "LocalSecondaryIndexes": [
            {
                "IndexName": "AlbumTitleIndex",
                "KeySchema": [
                    {
                        "AttributeName": "Artist",
                        "KeyType": "HASH"
                    },
                    {
                        "AttributeName": "AlbumTitle",
                        "KeyType": "RANGE"
                    }
                ],
                "Projection": {
                    "ProjectionType": "INCLUDE",
                    "NonKeyAttributes": [
                        "Genre",
                        "Year"
                    ]
                },
                "IndexSizeBytes": 0,
                "ItemCount": 0,
                "IndexArn": "arn:aws:dynamodb:us-west-2:123456789012:table/
MusicCollection/index/AlbumTitleIndex"
            }
        ]
    }
}
```

For more information, see Basic Operations for Tables in the *Amazon DynamoDB Developer Guide*.

Example 5: To create a table with a Global Secondary Index

The following example creates a table named `GameScores` with a Global Secondary Index called `GameTitleIndex`. The base table has a partition key of `UserId` and a sort key of `GameTitle`, allowing you to find an individual user's best score for a specific game efficiently, whereas the GSI has a partition key of `GameTitle` and a sort key of `TopScore`, allowing you to quickly find the overall highest score for a particular game.

```
aws dynamodb create-table \
    --table-name GameScores \
```

```
    --attribute-
definitions AttributeName=UserId,AttributeType=S AttributeName=GameTitle,AttributeType=S
 \
    --key-schema AttributeName=UserId,KeyType=HASH \
                 AttributeName=GameTitle,KeyType=RANGE \
    --provisioned-throughput ReadCapacityUnits=10,WriteCapacityUnits=5 \
    --global-secondary-indexes \
        "[
            {
                \"IndexName\": \"GameTitleIndex\",
                \"KeySchema\": [
                    {\"AttributeName\":\"GameTitle\",\"KeyType\":\"HASH\"},
                    {\"AttributeName\":\"TopScore\",\"KeyType\":\"RANGE\"}
                ],
                \"Projection\": {
                    \"ProjectionType\":\"INCLUDE\",
                    \"NonKeyAttributes\":[\"UserId\"]
                },
                \"ProvisionedThroughput\": {
                    \"ReadCapacityUnits\": 10,
                    \"WriteCapacityUnits\": 5
                }
            }
        ]"
```

Output:

```
{
    "TableDescription": {
        "AttributeDefinitions": [
            {
                "AttributeName": "GameTitle",
                "AttributeType": "S"
            },
            {
                "AttributeName": "TopScore",
                "AttributeType": "N"
            },
            {
                "AttributeName": "UserId",
                "AttributeType": "S"
            }
        ],
```

```
        "TableName": "GameScores",
        "KeySchema": [
            {
                "AttributeName": "UserId",
                "KeyType": "HASH"
            },
            {
                "AttributeName": "GameTitle",
                "KeyType": "RANGE"
            }
        ],
        "TableStatus": "CREATING",
        "CreationDateTime": "2020-05-26T17:28:15.602000-07:00",
        "ProvisionedThroughput": {
            "NumberOfDecreasesToday": 0,
            "ReadCapacityUnits": 10,
            "WriteCapacityUnits": 5
        },
        "TableSizeBytes": 0,
        "ItemCount": 0,
        "TableArn": "arn:aws:dynamodb:us-west-2:123456789012:table/GameScores",
        "TableId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111",
        "GlobalSecondaryIndexes": [
            {
                "IndexName": "GameTitleIndex",
                "KeySchema": [
                    {
                        "AttributeName": "GameTitle",
                        "KeyType": "HASH"
                    },
                    {
                        "AttributeName": "TopScore",
                        "KeyType": "RANGE"
                    }
                ],
                "Projection": {
                    "ProjectionType": "INCLUDE",
                    "NonKeyAttributes": [
                        "UserId"
                    ]
                },
                "IndexStatus": "CREATING",
                "ProvisionedThroughput": {
                    "NumberOfDecreasesToday": 0,
```

```
                    "ReadCapacityUnits": 10,
                    "WriteCapacityUnits": 5
                },
                "IndexSizeBytes": 0,
                "ItemCount": 0,
                "IndexArn": "arn:aws:dynamodb:us-west-2:123456789012:table/
GameScores/index/GameTitleIndex"
            }
        ]
    }
}
```

For more information, see Basic Operations for Tables in the *Amazon DynamoDB Developer Guide*.

Example 6: To create a table with multiple Global Secondary Indexes at once

The following example creates a table named `GameScores` with two Global Secondary Indexes. The GSI schemas are passed via a file, rather than on the command line.

```
aws dynamodb create-table \
    --table-name GameScores \
    --attribute-
definitions AttributeName=UserId,AttributeType=S AttributeName=GameTitle,AttributeType=S
 \
    --key-
schema AttributeName=UserId,KeyType=HASH AttributeName=GameTitle,KeyType=RANGE \
    --provisioned-throughput ReadCapacityUnits=10,WriteCapacityUnits=5 \
    --global-secondary-indexes file://gsi.json
```

Contents of `gsi.json`:

```
[
    {
        "IndexName": "GameTitleIndex",
        "KeySchema": [
            {
                "AttributeName": "GameTitle",
                "KeyType": "HASH"
            },
            {
                "AttributeName": "TopScore",
```

```
                "KeyType": "RANGE"
            }
        ],
        "Projection": {
            "ProjectionType": "ALL"
        },
        "ProvisionedThroughput": {
            "ReadCapacityUnits": 10,
            "WriteCapacityUnits": 5
        }
    },
    {
        "IndexName": "GameDateIndex",
        "KeySchema": [
            {
                "AttributeName": "GameTitle",
                "KeyType": "HASH"
            },
            {
                "AttributeName": "Date",
                "KeyType": "RANGE"
            }
        ],
        "Projection": {
            "ProjectionType": "ALL"
        },
        "ProvisionedThroughput": {
            "ReadCapacityUnits": 5,
            "WriteCapacityUnits": 5
        }
    }
]
```

Output:

```
{
    "TableDescription": {
        "AttributeDefinitions": [
            {
                "AttributeName": "Date",
                "AttributeType": "S"
            },
            {
```

```
                "AttributeName": "GameTitle",
                "AttributeType": "S"
            },
            {
                "AttributeName": "TopScore",
                "AttributeType": "N"
            },
            {
                "AttributeName": "UserId",
                "AttributeType": "S"
            }
        ],
        "TableName": "GameScores",
        "KeySchema": [
            {
                "AttributeName": "UserId",
                "KeyType": "HASH"
            },
            {
                "AttributeName": "GameTitle",
                "KeyType": "RANGE"
            }
        ],
        "TableStatus": "CREATING",
        "CreationDateTime": "2020-08-04T16:40:55.524000-07:00",
        "ProvisionedThroughput": {
            "NumberOfDecreasesToday": 0,
            "ReadCapacityUnits": 10,
            "WriteCapacityUnits": 5
        },
        "TableSizeBytes": 0,
        "ItemCount": 0,
        "TableArn": "arn:aws:dynamodb:us-west-2:123456789012:table/GameScores",
        "TableId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111",
        "GlobalSecondaryIndexes": [
            {
                "IndexName": "GameTitleIndex",
                "KeySchema": [
                    {
                        "AttributeName": "GameTitle",
                        "KeyType": "HASH"
                    },
                    {
                        "AttributeName": "TopScore",
```

```
                        "KeyType": "RANGE"
                    }
                ],
                "Projection": {
                    "ProjectionType": "ALL"
                },
                "IndexStatus": "CREATING",
                "ProvisionedThroughput": {
                    "NumberOfDecreasesToday": 0,
                    "ReadCapacityUnits": 10,
                    "WriteCapacityUnits": 5
                },
                "IndexSizeBytes": 0,
                "ItemCount": 0,
                "IndexArn": "arn:aws:dynamodb:us-west-2:123456789012:table/
GameScores/index/GameTitleIndex"
            },
            {
                "IndexName": "GameDateIndex",
                "KeySchema": [
                    {
                        "AttributeName": "GameTitle",
                        "KeyType": "HASH"
                    },
                    {
                        "AttributeName": "Date",
                        "KeyType": "RANGE"
                    }
                ],
                "Projection": {
                    "ProjectionType": "ALL"
                },
                "IndexStatus": "CREATING",
                "ProvisionedThroughput": {
                    "NumberOfDecreasesToday": 0,
                    "ReadCapacityUnits": 5,
                    "WriteCapacityUnits": 5
                },
                "IndexSizeBytes": 0,
                "ItemCount": 0,
                "IndexArn": "arn:aws:dynamodb:us-west-2:123456789012:table/
GameScores/index/GameDateIndex"
            }
        ]
```

```
    }
}
```

For more information, see Basic Operations for Tables in the *Amazon DynamoDB Developer Guide*.

Example 7: To create a table with Streams enabled

The following example creates a table called `GameScores` with DynamoDB Streams enabled. Both new and old images of each item will be written to the stream.

```
aws dynamodb create-table \
    --table-name GameScores \
    --attribute-
definitions AttributeName=UserId,AttributeType=S AttributeName=GameTitle,AttributeType=S
 \
    --key-
schema AttributeName=UserId,KeyType=HASH AttributeName=GameTitle,KeyType=RANGE \
    --provisioned-throughput ReadCapacityUnits=10,WriteCapacityUnits=5 \
    --stream-specification StreamEnabled=TRUE,StreamViewType=NEW_AND_OLD_IMAGES
```

Output:

```
{
    "TableDescription": {
        "AttributeDefinitions": [
            {
                "AttributeName": "GameTitle",
                "AttributeType": "S"
            },
            {
                "AttributeName": "UserId",
                "AttributeType": "S"
            }
        ],
        "TableName": "GameScores",
        "KeySchema": [
            {
                "AttributeName": "UserId",
                "KeyType": "HASH"
            },
            {
                "AttributeName": "GameTitle",
```

```
                "KeyType": "RANGE"
            }
        ],
        "TableStatus": "CREATING",
        "CreationDateTime": "2020-05-27T10:49:34.056000-07:00",
        "ProvisionedThroughput": {
            "NumberOfDecreasesToday": 0,
            "ReadCapacityUnits": 10,
            "WriteCapacityUnits": 5
        },
        "TableSizeBytes": 0,
        "ItemCount": 0,
        "TableArn": "arn:aws:dynamodb:us-west-2:123456789012:table/GameScores",
        "TableId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111",
        "StreamSpecification": {
            "StreamEnabled": true,
            "StreamViewType": "NEW_AND_OLD_IMAGES"
        },
        "LatestStreamLabel": "2020-05-27T17:49:34.056",
        "LatestStreamArn": "arn:aws:dynamodb:us-west-2:123456789012:table/
GameScores/stream/2020-05-27T17:49:34.056"
    }
}
```

For more information, see Basic Operations for Tables in the *Amazon DynamoDB Developer Guide*.

Example 8: To create a table with Keys-Only Stream enabled

The following example creates a table called `GameScores` with DynamoDB Streams enabled. Only the key attributes of modified items are written to the stream.

```
aws dynamodb create-table \
    --table-name GameScores \
    --attribute-
definitions AttributeName=UserId,AttributeType=S AttributeName=GameTitle,AttributeType=S
 \
    --key-
schema AttributeName=UserId,KeyType=HASH AttributeName=GameTitle,KeyType=RANGE \
    --provisioned-throughput ReadCapacityUnits=10,WriteCapacityUnits=5 \
    --stream-specification StreamEnabled=TRUE,StreamViewType=KEYS_ONLY
```

Output:

```
{
    "TableDescription": {
        "AttributeDefinitions": [
            {
                "AttributeName": "GameTitle",
                "AttributeType": "S"
            },
            {
                "AttributeName": "UserId",
                "AttributeType": "S"
            }
        ],
        "TableName": "GameScores",
        "KeySchema": [
            {
                "AttributeName": "UserId",
                "KeyType": "HASH"
            },
            {
                "AttributeName": "GameTitle",
                "KeyType": "RANGE"
            }
        ],
        "TableStatus": "CREATING",
        "CreationDateTime": "2023-05-25T18:45:34.140000+00:00",
        "ProvisionedThroughput": {
            "NumberOfDecreasesToday": 0,
            "ReadCapacityUnits": 10,
            "WriteCapacityUnits": 5
        },
        "TableSizeBytes": 0,
        "ItemCount": 0,
        "TableArn": "arn:aws:dynamodb:us-west-2:123456789012:table/GameScores",
        "TableId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111",
        "StreamSpecification": {
            "StreamEnabled": true,
            "StreamViewType": "KEYS_ONLY"
        },
        "LatestStreamLabel": "2023-05-25T18:45:34.140",
        "LatestStreamArn": "arn:aws:dynamodb:us-west-2:123456789012:table/
GameScores/stream/2023-05-25T18:45:34.140",
        "DeletionProtectionEnabled": false
    }
```

```
}
```

For more information, see Change data capture for DynamoDB Streams in the *Amazon DynamoDB Developer Guide*.

Example 9: To create a table with the Standard Infrequent Access class

The following example creates a table called `GameScores` and assigns the Standard-Infrequent Access (DynamoDB Standard-IA) table class. This table class is optimized for storage being the dominant cost.

```
aws dynamodb create-table \
    --table-name GameScores \
    --attribute-
definitions AttributeName=UserId,AttributeType=S AttributeName=GameTitle,AttributeType=S
 \
    --key-
schema AttributeName=UserId,KeyType=HASH AttributeName=GameTitle,KeyType=RANGE \
    --provisioned-throughput ReadCapacityUnits=10,WriteCapacityUnits=5 \
    --table-class STANDARD_INFREQUENT_ACCESS
```

Output:

```
{
    "TableDescription": {
        "AttributeDefinitions": [
            {
                "AttributeName": "GameTitle",
                "AttributeType": "S"
            },
            {
                "AttributeName": "UserId",
                "AttributeType": "S"
            }
        ],
        "TableName": "GameScores",
        "KeySchema": [
            {
                "AttributeName": "UserId",
                "KeyType": "HASH"
            },
            {
```

```
                "AttributeName": "GameTitle",
                "KeyType": "RANGE"
            }
        ],
        "TableStatus": "CREATING",
        "CreationDateTime": "2023-05-25T18:33:07.581000+00:00",
        "ProvisionedThroughput": {
            "NumberOfDecreasesToday": 0,
            "ReadCapacityUnits": 10,
            "WriteCapacityUnits": 5
        },
        "TableSizeBytes": 0,
        "ItemCount": 0,
        "TableArn": "arn:aws:dynamodb:us-west-2:123456789012:table/GameScores",
        "TableId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111",
        "TableClassSummary": {
            "TableClass": "STANDARD_INFREQUENT_ACCESS"
        },
        "DeletionProtectionEnabled": false
    }
}
```

For more information, see Table classes in the *Amazon DynamoDB Developer Guide*.

Example 10: To Create a table with Delete Protection enabled

The following example creates a table called `GameScores` and enables deletion protection.

```
aws dynamodb create-table \
    --table-name GameScores \
    --attribute-
definitions AttributeName=UserId,AttributeType=S AttributeName=GameTitle,AttributeType=S
 \
    --key-
schema AttributeName=UserId,KeyType=HASH AttributeName=GameTitle,KeyType=RANGE \
    --provisioned-throughput ReadCapacityUnits=10,WriteCapacityUnits=5 \
    --deletion-protection-enabled
```

Output:

```
{
    "TableDescription": {
```

```
        "AttributeDefinitions": [
            {
                "AttributeName": "GameTitle",
                "AttributeType": "S"
            },
            {
                "AttributeName": "UserId",
                "AttributeType": "S"
            }
        ],
        "TableName": "GameScores",
        "KeySchema": [
            {
                "AttributeName": "UserId",
                "KeyType": "HASH"
            },
            {
                "AttributeName": "GameTitle",
                "KeyType": "RANGE"
            }
        ],
        "TableStatus": "CREATING",
        "CreationDateTime": "2023-05-25T23:02:17.093000+00:00",
        "ProvisionedThroughput": {
            "NumberOfDecreasesToday": 0,
            "ReadCapacityUnits": 10,
            "WriteCapacityUnits": 5
        },
        "TableSizeBytes": 0,
        "ItemCount": 0,
        "TableArn": "arn:aws:dynamodb:us-west-2:123456789012:table/GameScores",
        "TableId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111",
        "DeletionProtectionEnabled": true
    }
}
```

For more information, see Using deletion protection in the *Amazon DynamoDB Developer Guide*.

- For API details, see CreateTable in *AWS CLI Command Reference*.

Go

SDK for Go V2

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import (
 "context"
 "errors"
 "log"
 "time"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/attributevalue"
 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/expression"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb/types"
)

// TableBasics encapsulates the Amazon DynamoDB service actions used in the
 examples.
// It contains a DynamoDB service client that is used to act on the specified
 table.
type TableBasics struct {
 DynamoDbClient *dynamodb.Client
 TableName      string
}



// CreateMovieTable creates a DynamoDB table with a composite primary key defined
 as
// a string sort key named `title`, and a numeric partition key named `year`.
// This function uses NewTableExistsWaiter to wait for the table to be created by
// DynamoDB before it returns.
func (basics TableBasics) CreateMovieTable(ctx context.Context)
 (*types.TableDescription, error) {
```

```
 var tableDesc *types.TableDescription
 table, err := basics.DynamoDbClient.CreateTable(ctx, &dynamodb.CreateTableInput{
  AttributeDefinitions: []types.AttributeDefinition{{
   AttributeName: aws.String("year"),
   AttributeType: types.ScalarAttributeTypeN,
  }, {
   AttributeName: aws.String("title"),
   AttributeType: types.ScalarAttributeTypeS,
  }},
  KeySchema: []types.KeySchemaElement{{
   AttributeName: aws.String("year"),
   KeyType:       types.KeyTypeHash,
  }, {
   AttributeName: aws.String("title"),
   KeyType:       types.KeyTypeRange,
  }},
  TableName:   aws.String(basics.TableName),
  BillingMode: types.BillingModePayPerRequest,
 })
 if err != nil {
  log.Printf("Couldn't create table %v. Here's why: %v\n", basics.TableName, err)
 } else {
  waiter := dynamodb.NewTableExistsWaiter(basics.DynamoDbClient)
  err = waiter.Wait(ctx, &dynamodb.DescribeTableInput{
   TableName: aws.String(basics.TableName)}, 5*time.Minute)
  if err != nil {
   log.Printf("Wait for table exists failed. Here's why: %v\n", err)
  }
  tableDesc = table.TableDescription
  log.Printf("Ccreating table test")
 }
 return tableDesc, err
}
```

- For API details, see CreateTable in *AWS SDK for Go API Reference*.

Java

SDK for Java 2.x

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import software.amazon.awssdk.core.waiters.WaiterResponse;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeDefinition;
import software.amazon.awssdk.services.dynamodb.model.BillingMode;
import software.amazon.awssdk.services.dynamodb.model.CreateTableRequest;
import software.amazon.awssdk.services.dynamodb.model.CreateTableResponse;
import software.amazon.awssdk.services.dynamodb.model.DescribeTableRequest;
import software.amazon.awssdk.services.dynamodb.model.DescribeTableResponse;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.KeySchemaElement;
import software.amazon.awssdk.services.dynamodb.model.KeyType;
import software.amazon.awssdk.services.dynamodb.model.OnDemandThroughput;
import software.amazon.awssdk.services.dynamodb.model.ProvisionedThroughput;
import software.amazon.awssdk.services.dynamodb.model.ScalarAttributeType;
import software.amazon.awssdk.services.dynamodb.waiters.DynamoDbWaiter;

/**
 * Before running this Java V2 code example, set up your development
 * environment, including your credentials.
 * <p>
 * For more information, see the following documentation topic:
 * <p>
 * https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-
started.html
 */
public class CreateTable {
    public static void main(String[] args) {
        final String usage = """

            Usage:
```

```
                <tableName> <key>

            Where:
                tableName - The Amazon DynamoDB table to create (for example,
Music3).
                key - The key for the Amazon DynamoDB table (for example,
Artist).
            """;

        if (args.length != 2) {
            System.out.println(usage);
            System.exit(1);
        }

        String tableName = args[0];
        String key = args[1];
        System.out.println("Creating an Amazon DynamoDB table " + tableName + "
with a simple primary key: " + key);
        Region region = Region.US_EAST_1;
        DynamoDbClient ddb = DynamoDbClient.builder()
            .region(region)
            .build();

        String result = createTable(ddb, tableName, key);
        System.out.println("New table is " + result);
        ddb.close();
    }

    public static String createTable(DynamoDbClient ddb, String tableName, String
key) {
        DynamoDbWaiter dbWaiter = ddb.waiter();
        CreateTableRequest request = CreateTableRequest.builder()
            .attributeDefinitions(AttributeDefinition.builder()
                .attributeName(key)
                .attributeType(ScalarAttributeType.S)
                .build())
            .keySchema(KeySchemaElement.builder()
                .attributeName(key)
                .keyType(KeyType.HASH)
                .build())
            .billingMode(BillingMode.PAY_PER_REQUEST) //  DynamoDB automatically
scales based on traffic.
            .tableName(tableName)
            .build();
```

```
        String newTable;
        try {
            CreateTableResponse response = ddb.createTable(request);
            DescribeTableRequest tableRequest = DescribeTableRequest.builder()
                .tableName(tableName)
                .build();

            // Wait until the Amazon DynamoDB table is created.
            WaiterResponse<DescribeTableResponse> waiterResponse =
 dbWaiter.waitUntilTableExists(tableRequest);
            waiterResponse.matched().response().ifPresent(System.out::println);
            newTable = response.tableDescription().tableName();
            return newTable;

        } catch (DynamoDbException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
        return "";
    }
}
```

- For API details, see CreateTable in *AWS SDK for Java 2.x API Reference*.

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import { CreateTableCommand, DynamoDBClient } from "@aws-sdk/client-dynamodb";

const client = new DynamoDBClient({});

export const main = async () => {
  const command = new CreateTableCommand({
```

```
    TableName: "EspressoDrinks",
    // For more information about data types,
    // see https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/
HowItWorks.NamingRulesDataTypes.html#HowItWorks.DataTypes and
    // https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/
Programming.LowLevelAPI.html#Programming.LowLevelAPI.DataTypeDescriptors
    AttributeDefinitions: [
      {
        AttributeName: "DrinkName",
        AttributeType: "S",
      },
    ],
    KeySchema: [
      {
        AttributeName: "DrinkName",
        KeyType: "HASH",
      },
    ],
    BillingMode: "PAY_PER_REQUEST",
  });

  const response = await client.send(command);
  console.log(response);
  return response;
};
```

- For more information, see AWS SDK for JavaScript Developer Guide.
- For API details, see CreateTable in *AWS SDK for JavaScript API Reference*.

SDK for JavaScript (v2)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });
```

```
// Create the DynamoDB service object
var ddb = new AWS.DynamoDB({ apiVersion: "2012-08-10" });

var params = {
  AttributeDefinitions: [
    {
      AttributeName: "CUSTOMER_ID",
      AttributeType: "N",
    },
    {
      AttributeName: "CUSTOMER_NAME",
      AttributeType: "S",
    },
  ],
  KeySchema: [
    {
      AttributeName: "CUSTOMER_ID",
      KeyType: "HASH",
    },
    {
      AttributeName: "CUSTOMER_NAME",
      KeyType: "RANGE",
    },
  ],
  ProvisionedThroughput: {
    ReadCapacityUnits: 1,
    WriteCapacityUnits: 1,
  },
  TableName: "CUSTOMER_LIST",
  StreamSpecification: {
    StreamEnabled: false,
  },
};

// Call DynamoDB to create the table
ddb.createTable(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Table Created", data);
  }
});
```

- For more information, see AWS SDK for JavaScript Developer Guide.
- For API details, see CreateTable in *AWS SDK for JavaScript API Reference*.

Kotlin

SDK for Kotlin

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
suspend fun createNewTable(
    tableNameVal: String,
    key: String,
): String? {
    val attDef =
        AttributeDefinition {
            attributeName = key
            attributeType = ScalarAttributeType.S
        }

    val keySchemaVal =
        KeySchemaElement {
            attributeName = key
            keyType = KeyType.Hash
        }

    val request =
        CreateTableRequest {
            attributeDefinitions = listOf(attDef)
            keySchema = listOf(keySchemaVal)
            billingMode = BillingMode.PayPerRequest
            tableName = tableNameVal
        }

    DynamoDbClient.fromEnvironment { region = "us-east-1" }.use { ddb ->
        var tableArn: String
```

```
        val response = ddb.createTable(request)
        ddb.waitUntilTableExists {
            // suspend call
            tableName = tableNameVal
        }
        tableArn = response.tableDescription!!.tableArn.toString()
        println("Table $tableArn is ready")
        return tableArn
    }
}
```

- For API details, see CreateTable in *AWS SDK for Kotlin API reference*.

PHP

SDK for PHP

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Create a table.

```
        $tableName = "ddb_demo_table_$uuid";
        $service->createTable(
            $tableName,
            [
                new DynamoDBAttribute('year', 'N', 'HASH'),
                new DynamoDBAttribute('title', 'S', 'RANGE')
            ]
        );

    public function createTable(string $tableName, array $attributes)
    {
        $keySchema = [];
        $attributeDefinitions = [];
        foreach ($attributes as $attribute) {
            if (is_a($attribute, DynamoDBAttribute::class)) {
```

```
                $keySchema[] = ['AttributeName' => $attribute->AttributeName,
'KeyType' => $attribute->KeyType];
                $attributeDefinitions[] =
                    ['AttributeName' => $attribute->AttributeName,
'AttributeType' => $attribute->AttributeType];
            }
        }

        $this->dynamoDbClient->createTable([
            'TableName' => $tableName,
            'KeySchema' => $keySchema,
            'AttributeDefinitions' => $attributeDefinitions,
            'ProvisionedThroughput' => ['ReadCapacityUnits' => 10,
'WriteCapacityUnits' => 10],
        ]);
    }
```

- For API details, see CreateTable in *AWS SDK for PHP API Reference*.

PowerShell

Tools for PowerShell V4

Example 1: This example creates a table named Thread that has a primary key consisting of 'ForumName' (key type hash) and 'Subject' (key type range). The schema used to construct the table can be piped into each cmdlet as shown or specified using the -Schema parameter.

```
$schema = New-DDBTableSchema
$schema | Add-DDBKeySchema -KeyName "ForumName" -KeyDataType "S"
$schema | Add-DDBKeySchema -KeyName "Subject" -KeyType RANGE -KeyDataType "S"
$schema | New-DDBTable -TableName "Thread" -ReadCapacity 10 -WriteCapacity 5
```

Output:

```
AttributeDefinitions   : {ForumName, Subject}
TableName              : Thread
KeySchema              : {ForumName, Subject}
TableStatus            : CREATING
CreationDateTime       : 10/28/2013 4:39:49 PM
ProvisionedThroughput  : Amazon.DynamoDBv2.Model.ProvisionedThroughputDescription
```

```
TableSizeBytes         : 0
ItemCount              : 0
LocalSecondaryIndexes  : {}
```

Example 2: This example creates a table named Thread that has a primary key consisting of 'ForumName' (key type hash) and 'Subject' (key type range). A local secondary index is also defined. The key of the local secondary index will be set automatically from the primary hash key on the table (ForumName). The schema used to construct the table can be piped into each cmdlet as shown or specified using the -Schema parameter.

```
$schema = New-DDBTableSchema
$schema | Add-DDBKeySchema -KeyName "ForumName" -KeyDataType "S"
$schema | Add-DDBKeySchema -KeyName "Subject" -KeyDataType "S"
$schema | Add-DDBIndexSchema -IndexName "LastPostIndex" -RangeKeyName
 "LastPostDateTime" -RangeKeyDataType "S" -ProjectionType "keys_only"
$schema | New-DDBTable -TableName "Thread" -ReadCapacity 10 -WriteCapacity 5
```

Output:

```
AttributeDefinitions   : {ForumName, LastPostDateTime, Subject}
TableName              : Thread
KeySchema              : {ForumName, Subject}
TableStatus            : CREATING
CreationDateTime       : 10/28/2013 4:39:49 PM
ProvisionedThroughput  : Amazon.DynamoDBv2.Model.ProvisionedThroughputDescription
TableSizeBytes         : 0
ItemCount              : 0
LocalSecondaryIndexes  : {LastPostIndex}
```

Example 3: This example shows how to use a single pipeline to create a table named Thread that has a primary key consisting of 'ForumName' (key type hash) and 'Subject' (key type range) and a local secondary index. The Add-DDBKeySchema and Add-DDBIndexSchema create a new TableSchema object for you if one is not supplied from the pipeline or the -Schema parameter.

```
New-DDBTableSchema |
  Add-DDBKeySchema -KeyName "ForumName" -KeyDataType "S" |
  Add-DDBKeySchema -KeyName "Subject" -KeyDataType "S" |
  Add-DDBIndexSchema -IndexName "LastPostIndex" `
                     -RangeKeyName "LastPostDateTime" `
```

```
                    -RangeKeyDataType "S" `
                    -ProjectionType "keys_only" |
  New-DDBTable -TableName "Thread" -ReadCapacity 10 -WriteCapacity 5
```

Output:

```
AttributeDefinitions   : {ForumName, LastPostDateTime, Subject}
TableName              : Thread
KeySchema              : {ForumName, Subject}
TableStatus            : CREATING
CreationDateTime       : 10/28/2013 4:39:49 PM
ProvisionedThroughput  : Amazon.DynamoDBv2.Model.ProvisionedThroughputDescription
TableSizeBytes         : 0
ItemCount              : 0
LocalSecondaryIndexes  : {LastPostIndex}
```

- For API details, see CreateTable in *AWS Tools for PowerShell Cmdlet Reference (V4)*.

Tools for PowerShell V5

Example 1: This example creates a table named Thread that has a primary key consisting of 'ForumName' (key type hash) and 'Subject' (key type range). The schema used to construct the table can be piped into each cmdlet as shown or specified using the -Schema parameter.

```
$schema = New-DDBTableSchema
$schema | Add-DDBKeySchema -KeyName "ForumName" -KeyDataType "S"
$schema | Add-DDBKeySchema -KeyName "Subject" -KeyType RANGE -KeyDataType "S"
$schema | New-DDBTable -TableName "Thread" -ReadCapacity 10 -WriteCapacity 5
```

Output:

```
AttributeDefinitions   : {ForumName, Subject}
TableName              : Thread
KeySchema              : {ForumName, Subject}
TableStatus            : CREATING
CreationDateTime       : 10/28/2013 4:39:49 PM
ProvisionedThroughput  : Amazon.DynamoDBv2.Model.ProvisionedThroughputDescription
TableSizeBytes         : 0
ItemCount              : 0
LocalSecondaryIndexes  : {}
```

Example 2: This example creates a table named Thread that has a primary key consisting of 'ForumName' (key type hash) and 'Subject' (key type range). A local secondary index is also defined. The key of the local secondary index will be set automatically from the primary hash key on the table (ForumName). The schema used to construct the table can be piped into each cmdlet as shown or specified using the -Schema parameter.

```
$schema = New-DDBTableSchema
$schema | Add-DDBKeySchema -KeyName "ForumName" -KeyDataType "S"
$schema | Add-DDBKeySchema -KeyName "Subject" -KeyDataType "S"
$schema | Add-DDBIndexSchema -IndexName "LastPostIndex" -RangeKeyName
 "LastPostDateTime" -RangeKeyDataType "S" -ProjectionType "keys_only"
$schema | New-DDBTable -TableName "Thread" -ReadCapacity 10 -WriteCapacity 5
```

Output:

```
AttributeDefinitions   : {ForumName, LastPostDateTime, Subject}
TableName              : Thread
KeySchema              : {ForumName, Subject}
TableStatus            : CREATING
CreationDateTime       : 10/28/2013 4:39:49 PM
ProvisionedThroughput  : Amazon.DynamoDBv2.Model.ProvisionedThroughputDescription
TableSizeBytes         : 0
ItemCount              : 0
LocalSecondaryIndexes  : {LastPostIndex}
```

Example 3: This example shows how to use a single pipeline to create a table named Thread that has a primary key consisting of 'ForumName' (key type hash) and 'Subject' (key type range) and a local secondary index. The Add-DDBKeySchema and Add-DDBIndexSchema create a new TableSchema object for you if one is not supplied from the pipeline or the -Schema parameter.

```
New-DDBTableSchema |
  Add-DDBKeySchema -KeyName "ForumName" -KeyDataType "S" |
  Add-DDBKeySchema -KeyName "Subject" -KeyDataType "S" |
  Add-DDBIndexSchema -IndexName "LastPostIndex" `
                     -RangeKeyName "LastPostDateTime" `
                     -RangeKeyDataType "S" `
                     -ProjectionType "keys_only" |
  New-DDBTable -TableName "Thread" -ReadCapacity 10 -WriteCapacity 5
```

Output:

```
AttributeDefinitions    : {ForumName, LastPostDateTime, Subject}
TableName               : Thread
KeySchema               : {ForumName, Subject}
TableStatus             : CREATING
CreationDateTime        : 10/28/2013 4:39:49 PM
ProvisionedThroughput   : Amazon.DynamoDBv2.Model.ProvisionedThroughputDescription
TableSizeBytes          : 0
ItemCount               : 0
LocalSecondaryIndexes   : {LastPostIndex}
```

- For API details, see CreateTable in *AWS Tools for PowerShell Cmdlet Reference (V5)*.

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Create a table for storing movie data.

```
class Movies:
    """Encapsulates an Amazon DynamoDB table of movie data.

    Example data structure for a movie record in this table:
        {
            "year": 1999,
            "title": "For Love of the Game",
            "info": {
                "directors": ["Sam Raimi"],
                "release_date": "1999-09-15T00:00:00Z",
                "rating": 6.3,
                "plot": "A washed up pitcher flashes through his career.",
                "rank": 4987,
                "running_time_secs": 8220,
                "actors": [
                    "Kevin Costner",
                    "Kelly Preston",
```

```
                    "John C. Reilly"
                ]
            }
        }
    """

    def __init__(self, dyn_resource):
        """
        :param dyn_resource: A Boto3 DynamoDB resource.
        """
        self.dyn_resource = dyn_resource
        # The table variable is set during the scenario in the call to
        # 'exists' if the table exists. Otherwise, it is set by 'create_table'.
        self.table = None


    def create_table(self, table_name):
        """
        Creates an Amazon DynamoDB table that can be used to store movie data.
        The table uses the release year of the movie as the partition key and the
        title as the sort key.

        :param table_name: The name of the table to create.
        :return: The newly created table.
        """
        try:
            self.table = self.dyn_resource.create_table(
                TableName=table_name,
                KeySchema=[
                    {"AttributeName": "year", "KeyType": "HASH"},  # Partition
key
                    {"AttributeName": "title", "KeyType": "RANGE"},  # Sort key
                ],
                AttributeDefinitions=[
                    {"AttributeName": "year", "AttributeType": "N"},
                    {"AttributeName": "title", "AttributeType": "S"},
                ],
                BillingMode='PAY_PER_REQUEST',
            )
            self.table.wait_until_exists()
        except ClientError as err:
            logger.error(
                "Couldn't create table %s. Here's why: %s: %s",
                table_name,
```

```
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
        else:
            return self.table
```

- For API details, see CreateTable in *AWS SDK for Python (Boto3) API Reference*.

Ruby

SDK for Ruby

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
# Encapsulates an Amazon DynamoDB table of movie data.
class Scaffold
  attr_reader :dynamo_resource, :table_name, :table

  def initialize(table_name)
    client = Aws::DynamoDB::Client.new(region: 'us-east-1')
    @dynamo_resource = Aws::DynamoDB::Resource.new(client: client)
    @table_name = table_name
    @table = nil
    @logger = Logger.new($stdout)
    @logger.level = Logger::DEBUG
  end

  # Creates an Amazon DynamoDB table that can be used to store movie data.
  # The table uses the release year of the movie as the partition key and the
  # title as the sort key.
  #
  # @param table_name [String] The name of the table to create.
  # @return [Aws::DynamoDB::Table] The newly created table.
  def create_table(table_name)
    @table = @dynamo_resource.create_table(
```

```
      table_name: table_name,
      key_schema: [
        { attribute_name: 'year', key_type: 'HASH' }, # Partition key
        { attribute_name: 'title', key_type: 'RANGE' } # Sort key
      ],
      attribute_definitions: [
        { attribute_name: 'year', attribute_type: 'N' },
        { attribute_name: 'title', attribute_type: 'S' }
      ],
      billing_mode: 'PAY_PER_REQUEST'
    )
    @dynamo_resource.client.wait_until(:table_exists, table_name: table_name)
    @table
  rescue Aws::DynamoDB::Errors::ServiceError => e
    @logger.error("Failed create table #{table_name}:\n#{e.code}: #{e.message}")
    raise
  end
```

- For API details, see CreateTable in *AWS SDK for Ruby API Reference*.

Rust

SDK for Rust

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
pub async fn create_table(
    client: &Client,
    table: &str,
    key: &str,
) -> Result<CreateTableOutput, Error> {
    let a_name: String = key.into();
    let table_name: String = table.into();

    let ad = AttributeDefinition::builder()
        .attribute_name(&a_name)
        .attribute_type(ScalarAttributeType::S)
```

```
        .build()
        .map_err(Error::BuildError)?;

    let ks = KeySchemaElement::builder()
        .attribute_name(&a_name)
        .key_type(KeyType::Hash)
        .build()
        .map_err(Error::BuildError)?;

    let create_table_response = client
        .create_table()
        .table_name(table_name)
        .key_schema(ks)
        .attribute_definitions(ad)
        .billing_mode(BillingMode::PayPerRequest)
        .send()
        .await;

    match create_table_response {
        Ok(out) => {
            println!("Added table {} with key {}", table, key);
            Ok(out)
        }
        Err(e) => {
            eprintln!("Got an error creating table:");
            eprintln!("{}", e);
            Err(Error::unhandled(e))
        }
    }
}
```

- For API details, see CreateTable in *AWS SDK for Rust API reference*.

SAP ABAP

SDK for SAP ABAP

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    TRY.
        DATA(lt_keyschema) = VALUE /aws1/cl_dynkeyschemaelement=>tt_keyschema(
          ( NEW /aws1/cl_dynkeyschemaelement( iv_attributename = 'year'
                                              iv_keytype = 'HASH' ) )
          ( NEW /aws1/cl_dynkeyschemaelement( iv_attributename = 'title'
                                              iv_keytype = 'RANGE' ) ) ).
        DATA(lt_attributedefinitions) = VALUE /aws1/
cl_dynattributedefn=>tt_attributedefinitions(
          ( NEW /aws1/cl_dynattributedefn( iv_attributename = 'year'
                                           iv_attributetype = 'N' ) )
          ( NEW /aws1/cl_dynattributedefn( iv_attributename = 'title'
                                           iv_attributetype = 'S' ) ) ).

        " Adjust read/write capacities as desired.
        DATA(lo_dynprovthroughput)  = NEW /aws1/cl_dynprovthroughput(
          iv_readcapacityunits = 5
          iv_writecapacityunits = 5 ).
        oo_result = lo_dyn->createtable(
          it_keyschema = lt_keyschema
          iv_tablename = iv_table_name
          it_attributedefinitions = lt_attributedefinitions
          io_provisionedthroughput = lo_dynprovthroughput ).
        " Table creation can take some time. Wait till table exists before
 returning.
        lo_dyn->get_waiter( )->tableexists(
          iv_max_wait_time = 200
          iv_tablename     = iv_table_name ).
        MESSAGE 'DynamoDB Table' && iv_table_name && 'created.' TYPE 'I'.
        " This exception can happen if the table already exists.
      CATCH /aws1/cx_dynresourceinuseex INTO DATA(lo_resourceinuseex).
        DATA(lv_error) = |"{ lo_resourceinuseex->av_err_code }" -
{ lo_resourceinuseex->av_err_msg }|.
        MESSAGE lv_error TYPE 'E'.
    ENDTRY.
```

- For API details, see CreateTable in *AWS SDK for SAP ABAP API reference*.

Swift

SDK for Swift

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import AWSDynamoDB


    ///
    /// Create a movie table in the Amazon DynamoDB data store.
    ///
    private func createTable() async throws {
        do {
            guard let client = self.ddbClient else {
                throw MoviesError.UninitializedClient
            }

            let input = CreateTableInput(
                attributeDefinitions: [
                    DynamoDBClientTypes.AttributeDefinition(attributeName:
 "year", attributeType: .n),
                    DynamoDBClientTypes.AttributeDefinition(attributeName:
 "title", attributeType: .s)
                ],
                billingMode: DynamoDBClientTypes.BillingMode.payPerRequest,
                keySchema: [
                    DynamoDBClientTypes.KeySchemaElement(attributeName: "year",
 keyType: .hash),
                    DynamoDBClientTypes.KeySchemaElement(attributeName: "title",
 keyType: .range)
                ],
                tableName: self.tableName
            )
            let output = try await client.createTable(input: input)
            if output.tableDescription == nil {
                throw MoviesError.TableNotFound
            }
```

```
        } catch {
            print("ERROR: createTable:", dump(error))
            throw error
        }
    }
```

- For API details, see CreateTable in *AWS SDK for Swift API reference*.

For more DynamoDB examples, see Code examples for DynamoDB using AWS SDKs.

After creating the new table, proceed to Step 2: Write data to a DynamoDB table.

Step 2: Write data to a DynamoDB table

In this step, you insert several items into the `Music` table that you created in Step 1: Create a table in DynamoDB.

For more information about write operations, see Writing an item.

AWS Management Console

Follow these steps to write data to the `Music` table using the DynamoDB console.

1. Open the DynamoDB console at https://console.aws.amazon.com/dynamodb/.
2. In the left navigation pane, choose **Tables**.
3. On the **Tables** page, choose the **Music** table.
4. Choose **Explore table items**.
5. In the **Items returned** section, choose **Create item**.
6. On the **Create item** page, do the following to add items to your table:
    a. Choose **Add new attribute**, and then choose **Number**.
    b. For Attribute name, enter **`Awards`**.
    c. Repeat this process to create an **`AlbumTitle`** of type **String**.
    d. Enter the following values for your item:
        i. For **Artist**, enter **`No One You Know`**.

ii. For **SongTitle**, enter **`Call Me Today`**.

iii. For **AlbumTitle**, enter **`Somewhat Famous`**.

iv. For **Awards**, enter **1**.

7. Choose **Create item**.
8. Repeat this process and create another item with the following values:

   a. For **Artist**, enter **`Acme Band`**.

   b. For **SongTitle** enter **`Happy Day`**.

   c. For **AlbumTitle**, enter **`Songs About Life`**.

   d. For **Awards**, enter **`10`**.

9. Do this one more time to create another item with the same **Artist** as the previous step, but different values for the other attributes:

   a. For **Artist**, enter **`Acme Band`**.

   b. For **SongTitle** enter **`PartiQL Rocks`**.

   c. For **AlbumTitle**, enter **`Another Album Title`**.

   d. For **Awards**, enter **8**.

AWS CLI

The following AWS CLI example creates several new items in the `Music` table. You can do this either through the DynamoDB API or PartiQL, a SQL-compatible query language for DynamoDB.

DynamoDB API

Linux

```
aws dynamodb put-item \
    --table-name Music  \
    --item \
        '{"Artist": {"S": "No One You Know"}, "SongTitle": {"S": "Call Me Today"},
 "AlbumTitle": {"S": "Somewhat Famous"}, "Awards": {"N": "1"}}'

aws dynamodb put-item \
    --table-name Music  \
    --item \
```

```
        '{"Artist": {"S": "No One You Know"}, "SongTitle": {"S": "Howdy"},
 "AlbumTitle": {"S": "Somewhat Famous"}, "Awards": {"N": "2"}}'

aws dynamodb put-item \
    --table-name Music \
    --item \
        '{"Artist": {"S": "Acme Band"}, "SongTitle": {"S": "Happy Day"},
 "AlbumTitle": {"S": "Songs About Life"}, "Awards": {"N": "10"}}'

aws dynamodb put-item \
    --table-name Music \
    --item \
        '{"Artist": {"S": "Acme Band"}, "SongTitle": {"S": "PartiQL Rocks"},
 "AlbumTitle": {"S": "Another Album Title"}, "Awards": {"N": "8"}}'
```

Windows CMD

```
aws dynamodb put-item ^
    --table-name Music  ^
    --item ^
        "{\"Artist\": {\"S\": \"No One You Know\"}, \"SongTitle\": {\"S\": \"Call
 Me Today\"}, \"AlbumTitle\": {\"S\": \"Somewhat Famous\"}, \"Awards\": {\"N\":
 \"1\"}}"

aws dynamodb put-item ^
    --table-name Music  ^
    --item ^
        "{\"Artist\": {\"S\": \"No One You Know\"}, \"SongTitle\": {\"S\": \"Howdy
\"}, \"AlbumTitle\": {\"S\": \"Somewhat Famous\"}, \"Awards\": {\"N\": \"2\"}}"

aws dynamodb put-item ^
    --table-name Music ^
    --item ^
        "{\"Artist\": {\"S\": \"Acme Band\"}, \"SongTitle\": {\"S\": \"Happy Day\"},
 \"AlbumTitle\": {\"S\": \"Songs About Life\"}, \"Awards\": {\"N\": \"10\"}}"

aws dynamodb put-item ^
    --table-name Music ^
    --item ^
        "{\"Artist\": {\"S\": \"Acme Band\"}, \"SongTitle\": {\"S\": \"PartiQL Rocks
\"}, \"AlbumTitle\": {\"S\": \"Another Album Title\"}, \"Awards\": {\"N\": \"8\"}}"
```

Linux

```
aws dynamodb execute-statement --statement "INSERT INTO Music  \
                VALUE  \
                {'Artist':'No One You Know','SongTitle':'Call Me Today',
 'AlbumTitle':'Somewhat Famous', 'Awards':'1'}"

aws dynamodb execute-statement --statement "INSERT INTO Music  \
                VALUE  \
                {'Artist':'No One You Know','SongTitle':'Howdy',
 'AlbumTitle':'Somewhat Famous', 'Awards':'2'}"

aws dynamodb execute-statement --statement "INSERT INTO Music  \
                VALUE  \
                {'Artist':'Acme Band','SongTitle':'Happy Day', 'AlbumTitle':'Songs
 About Life', 'Awards':'10'}"

aws dynamodb execute-statement --statement "INSERT INTO Music  \
                VALUE  \
                {'Artist':'Acme Band','SongTitle':'PartiQL Rocks',
 'AlbumTitle':'Another Album Title', 'Awards':'8'}"
```

Windows CMD

```
aws dynamodb execute-statement --statement "INSERT INTO Music VALUE {'Artist':'No
 One You Know','SongTitle':'Call Me Today', 'AlbumTitle':'Somewhat Famous',
 'Awards':'1'}"

aws dynamodb execute-statement --statement "INSERT INTO Music VALUE {'Artist':'No
 One You Know','SongTitle':'Howdy', 'AlbumTitle':'Somewhat Famous', 'Awards':'2'}"

aws dynamodb execute-statement --statement "INSERT INTO Music VALUE {'Artist':'Acme
 Band','SongTitle':'Happy Day', 'AlbumTitle':'Songs About Life', 'Awards':'10'}"

aws dynamodb execute-statement --statement "INSERT INTO Music VALUE {'Artist':'Acme
 Band','SongTitle':'PartiQL Rocks', 'AlbumTitle':'Another Album Title',
 'Awards':'8'}"
```

For more information about writing data with PartiQL, see PartiQL insert statements.

For more information about supported data types in DynamoDB, see Data types.

For more information about how to represent DynamoDB data types in JSON, see Attribute values.

AWS SDK

The following code examples show how to write an item to a DynamoDB table using an AWS SDK.

.NET

SDK for .NET (v4)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    /// <summary>
    /// Adds a new item to the table.
    /// </summary>
    /// <param name="newMovie">A Movie object containing informtation for
    /// the movie to add to the table.</param>
    /// <param name="tableName">The name of the table where the item will be
 added.</param>
    /// <returns>A Boolean value that indicates the results of adding the item.</
returns>
    public async Task<bool> PutItemAsync(Movie newMovie, string tableName)
    {
        try
        {
            var item = new Dictionary<string, AttributeValue>
            {
                ["title"] = new AttributeValue { S = newMovie.Title },
                ["year"] = new AttributeValue { N = newMovie.Year.ToString() },
            };

            var request = new PutItemRequest
            {
                TableName = tableName,
                Item = item,
            };
```

```
            await _amazonDynamoDB.PutItemAsync(request);
            return true;
        }
        catch (ResourceNotFoundException ex)
        {
            Console.WriteLine($"Table {tableName} was not found. {ex.Message}");
            return false;
        }
        catch (AmazonDynamoDBException ex)
        {
            Console.WriteLine($"An Amazon DynamoDB error occurred while putting
item. {ex.Message}");
            throw;
        }
        catch (Exception ex)
        {
            Console.WriteLine($"An error occurred while putting item.
{ex.Message}");
            throw;
        }
    }
```

- For API details, see PutItem in *AWS SDK for .NET API Reference*.

Bash

AWS CLI with Bash script

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
###############################################################################
# function dynamodb_put_item
#
# This function puts an item into a DynamoDB table.
#
```

```
# Parameters:
#       -n table_name  -- The name of the table.
#       -i item  -- Path to json file containing the item values.
#
#  Returns:
#       0 - If successful.
#       1 - If it fails.
###############################################################################
function dynamodb_put_item() {
  local table_name item response
  local option OPTARG # Required to use getopts command in a function.

  ######################################
  # Function usage explanation
  ######################################
  function usage() {
    echo "function dynamodb_put_item"
    echo "Put an item into a DynamoDB table."
    echo " -n table_name  -- The name of the table."
    echo " -i item  -- Path to json file containing the item values."
    echo ""
  }

  while getopts "n:i:h" option; do
    case "${option}" in
      n) table_name="${OPTARG}" ;;
      i) item="${OPTARG}" ;;
      h)
        usage
        return 0
        ;;
      \?)
        echo "Invalid parameter"
        usage
        return 1
        ;;
    esac
  done
  export OPTIND=1

  if [[ -z "$table_name" ]]; then
    errecho "ERROR: You must provide a table name with the -n parameter."
    usage
    return 1
```

```
  fi

  if [[ -z "$item" ]]; then
    errecho "ERROR: You must provide an item with the -i parameter."
    usage
    return 1
  fi

  iecho "Parameters:\n"
  iecho "    table_name:   $table_name"
  iecho "    item:   $item"
  iecho ""
  iecho ""

  response=$(aws dynamodb put-item \
    --table-name "$table_name" \
    --item file://"$item")

  local error_code=${?}

  if [[ $error_code -ne 0 ]]; then
    aws_cli_error_log $error_code
    errecho "ERROR: AWS reports put-item operation failed.$response"
    return 1
  fi

  return 0

}
```

The utility functions used in this example.

```
###############################################################################
# function iecho
#
# This function enables the script to display the specified text only if
# the global variable $VERBOSE is set to true.
###############################################################################
function iecho() {
  if [[ $VERBOSE == true ]]; then
    echo "$@"
  fi
```

```
}

##############################################################################
# function errecho
#
# This function outputs everything sent to it to STDERR (standard error output).
##############################################################################
function errecho() {
  printf "%s\n" "$*" 1>&2
}

##############################################################################
# function aws_cli_error_log()
#
# This function is used to log the error messages from the AWS CLI.
#
# See https://docs.aws.amazon.com/cli/latest/topic/return-codes.html#cli-aws-
help-return-codes.
#
# The function expects the following argument:
#         $1 - The error code returned by the AWS CLI.
#
#  Returns:
#          0: - Success.
#
##############################################################################
function aws_cli_error_log() {
  local err_code=$1
  errecho "Error code : $err_code"
  if [ "$err_code" == 1 ]; then
    errecho "  One or more S3 transfers failed."
  elif [ "$err_code" == 2 ]; then
    errecho "  Command line failed to parse."
  elif [ "$err_code" == 130 ]; then
    errecho "  Process received SIGINT."
  elif [ "$err_code" == 252 ]; then
    errecho "  Command syntax invalid."
  elif [ "$err_code" == 253 ]; then
    errecho "  The system environment or configuration was invalid."
  elif [ "$err_code" == 254 ]; then
    errecho "  The service returned an error."
  elif [ "$err_code" == 255 ]; then
    errecho "  255 is a catch-all error."
  fi
```

```
  return 0
}
```

- For API details, see PutItem in *AWS CLI Command Reference*.

C++

SDK for C++

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
//! Put an item in an Amazon DynamoDB table.
/*!
  \sa putItem()
  \param tableName: The table name.
  \param artistKey: The artist key. This is the partition key for the table.
  \param artistValue: The artist value.
  \param albumTitleKey: The album title key.
  \param albumTitleValue: The album title value.
  \param awardsKey: The awards key.
  \param awardsValue: The awards value.
  \param songTitleKey: The song title key.
  \param songTitleValue: The song title value.
  \param clientConfiguration: AWS client configuration.
  \return bool: Function succeeded.
 */
bool AwsDoc::DynamoDB::putItem(const Aws::String &tableName,
                               const Aws::String &artistKey,
                               const Aws::String &artistValue,
                               const Aws::String &albumTitleKey,
                               const Aws::String &albumTitleValue,
                               const Aws::String &awardsKey,
                               const Aws::String &awardsValue,
                               const Aws::String &songTitleKey,
                               const Aws::String &songTitleValue,
```

```
                                    const Aws::Client::ClientConfiguration
 &clientConfiguration) {
    Aws::DynamoDB::DynamoDBClient dynamoClient(clientConfiguration);

    Aws::DynamoDB::Model::PutItemRequest putItemRequest;
    putItemRequest.SetTableName(tableName);

    putItemRequest.AddItem(artistKey,
 Aws::DynamoDB::Model::AttributeValue().SetS(
            artistValue)); // This is the hash key.
    putItemRequest.AddItem(albumTitleKey,
 Aws::DynamoDB::Model::AttributeValue().SetS(
            albumTitleValue));
    putItemRequest.AddItem(awardsKey,

 Aws::DynamoDB::Model::AttributeValue().SetS(awardsValue));
    putItemRequest.AddItem(songTitleKey,

 Aws::DynamoDB::Model::AttributeValue().SetS(songTitleValue));

    const Aws::DynamoDB::Model::PutItemOutcome outcome = dynamoClient.PutItem(
            putItemRequest);
    if (outcome.IsSuccess()) {
        std::cout << "Successfully added Item!" << std::endl;
    }
    else {
        std::cerr << outcome.GetError().GetMessage() << std::endl;
        return false;
    }

    return waitTableActive(tableName, dynamoClient);
}
```

Code that waits for the table to become active.

```
//! Query a newly created DynamoDB table until it is active.
/*!
  \sa waitTableActive()
  \param waitTableActive: The DynamoDB table's name.
  \param dynamoClient: A DynamoDB client.
  \return bool: Function succeeded.
```

```
*/
bool AwsDoc::DynamoDB::waitTableActive(const Aws::String &tableName,
                                       const Aws::DynamoDB::DynamoDBClient
 &dynamoClient) {

    // Repeatedly call DescribeTable until table is ACTIVE.
    const int MAX_QUERIES = 20;
    Aws::DynamoDB::Model::DescribeTableRequest request;
    request.SetTableName(tableName);

    int count = 0;
    while (count < MAX_QUERIES) {
        const Aws::DynamoDB::Model::DescribeTableOutcome &result =
 dynamoClient.DescribeTable(
                request);
        if (result.IsSuccess()) {
            Aws::DynamoDB::Model::TableStatus status =
 result.GetResult().GetTable().GetTableStatus();

            if (Aws::DynamoDB::Model::TableStatus::ACTIVE != status) {
                std::this_thread::sleep_for(std::chrono::seconds(1));
            }
            else {
                return true;
            }
        }
        else {
            std::cerr << "Error DynamoDB::waitTableActive "
                      << result.GetError().GetMessage() << std::endl;
            return false;
        }
        count++;
    }
    return false;
}
```

- For API details, see PutItem in *AWS SDK for C++ API Reference*.

CLI

AWS CLI

Example 1: To add an item to a table

The following `put-item` example adds a new item to the *MusicCollection* table.

```
aws dynamodb put-item \
    --table-name MusicCollection \
    --item file://item.json \
    --return-consumed-capacity TOTAL \
    --return-item-collection-metrics SIZE
```

Contents of `item.json`:

```
{
    "Artist": {"S": "No One You Know"},
    "SongTitle": {"S": "Call Me Today"},
    "AlbumTitle": {"S": "Greatest Hits"}
}
```

Output:

```
{
    "ConsumedCapacity": {
        "TableName": "MusicCollection",
        "CapacityUnits": 1.0
    },
    "ItemCollectionMetrics": {
        "ItemCollectionKey": {
            "Artist": {
                "S": "No One You Know"
            }
        },
        "SizeEstimateRangeGB": [
            0.0,
            1.0
        ]
    }
}
```

For more information, see Writing an Item in the *Amazon DynamoDB Developer Guide*.

Example 2: To conditionally overwrite an item in a table

The following `put-item` example overwrites an existing item in the `MusicCollection` table only if that existing item has an `AlbumTitle` attribute with a value of `Greatest Hits`. The command returns the previous value of the item.

```
aws dynamodb put-item \
    --table-name MusicCollection \
    --item file://item.json \
    --condition-expression "#A = :A" \
    --expression-attribute-names file://names.json \
    --expression-attribute-values file://values.json \
    --return-values ALL_OLD
```

Contents of `item.json`:

```
{
    "Artist": {"S": "No One You Know"},
    "SongTitle": {"S": "Call Me Today"},
    "AlbumTitle": {"S": "Somewhat Famous"}
}
```

Contents of `names.json`:

```
{
    "#A": "AlbumTitle"
}
```

Contents of `values.json`:

```
{
    ":A": {"S": "Greatest Hits"}
}
```

Output:

```
{
    "Attributes": {
```

```
        "AlbumTitle": {
            "S": "Greatest Hits"
        },
        "Artist": {
            "S": "No One You Know"
        },
        "SongTitle": {
            "S": "Call Me Today"
        }
    }
}
```

If the key already exists, you should see the following output:

```
A client error (ConditionalCheckFailedException) occurred when calling the
 PutItem operation: The conditional request failed.
```

For more information, see Writing an Item in the *Amazon DynamoDB Developer Guide*.

- For API details, see PutItem in *AWS CLI Command Reference*.

Go

SDK for Go V2

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import (
 "context"
 "errors"
 "log"
 "time"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/attributevalue"
 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/expression"
```

```
 "github.com/aws/aws-sdk-go-v2/service/dynamodb"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb/types"
)

// TableBasics encapsulates the Amazon DynamoDB service actions used in the
 examples.
// It contains a DynamoDB service client that is used to act on the specified
 table.
type TableBasics struct {
 DynamoDbClient *dynamodb.Client
 TableName      string
}



// AddMovie adds a movie the DynamoDB table.
func (basics TableBasics) AddMovie(ctx context.Context, movie Movie) error {
 item, err := attributevalue.MarshalMap(movie)
 if err != nil {
  panic(err)
 }
 _, err = basics.DynamoDbClient.PutItem(ctx, &dynamodb.PutItemInput{
  TableName: aws.String(basics.TableName), Item: item,
 })
 if err != nil {
  log.Printf("Couldn't add item to table. Here's why: %v\n", err)
 }
 return err
}
```

Define a Movie struct that is used in this example.

```
import (
 "archive/zip"
 "bytes"
 "encoding/json"
 "fmt"
 "io"
 "log"
 "net/http"
```

```
 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/attributevalue"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb/types"
)

// Movie encapsulates data about a movie. Title and Year are the composite
 primary key
// of the movie in Amazon DynamoDB. Title is the sort key, Year is the partition
 key,
// and Info is additional data.
type Movie struct {
 Title string                 `dynamodbav:"title"`
 Year  int                    `dynamodbav:"year"`
 Info  map[string]interface{} `dynamodbav:"info"`
}

// GetKey returns the composite primary key of the movie in a format that can be
// sent to DynamoDB.
func (movie Movie) GetKey() map[string]types.AttributeValue {
 title, err := attributevalue.Marshal(movie.Title)
 if err != nil {
  panic(err)
 }
 year, err := attributevalue.Marshal(movie.Year)
 if err != nil {
  panic(err)
 }
 return map[string]types.AttributeValue{"title": title, "year": year}
}

// String returns the title, year, rating, and plot of a movie, formatted for the
 example.
func (movie Movie) String() string {
 return fmt.Sprintf("%v\n\tReleased: %v\n\tRating: %v\n\tPlot: %v\n",
  movie.Title, movie.Year, movie.Info["rating"], movie.Info["plot"])
}
```

- For API details, see PutItem in *AWS SDK for Go API Reference*.

Java

SDK for Java 2.x

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Puts an item into a table using DynamoDbClient.

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.PutItemRequest;
import software.amazon.awssdk.services.dynamodb.model.PutItemResponse;
import software.amazon.awssdk.services.dynamodb.model.ResourceNotFoundException;
import java.util.HashMap;

/**
 * Before running this Java V2 code example, set up your development
 * environment, including your credentials.
 *
 * For more information, see the following documentation topic:
 *
 * https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-
started.html
 *
 * To place items into an Amazon DynamoDB table using the AWS SDK for Java V2,
 * its better practice to use the
 * Enhanced Client. See the EnhancedPutItem example.
 */
public class PutItem {
    public static void main(String[] args) {
        final String usage = """

                Usage:
                    <tableName> <key> <keyVal> <albumtitle> <albumtitleval>
 <awards> <awardsval> <Songtitle> <songtitleval>

                Where:
```

```
                    tableName - The Amazon DynamoDB table in which an item is
placed (for example, Music3).
                    key - The key used in the Amazon DynamoDB table (for example,
Artist).
                    keyval - The key value that represents the item to get (for
example, Famous Band).
                    albumTitle - The Album title (for example, AlbumTitle).
                    AlbumTitleValue - The name of the album (for example, Songs
About Life ).
                    Awards - The awards column (for example, Awards).
                    AwardVal - The value of the awards (for example, 10).
                    SongTitle - The song title (for example, SongTitle).
                    SongTitleVal - The value of the song title (for example,
Happy Day).
                **Warning** This program will  place an item that you specify
into a table!
                """;

        if (args.length != 9) {
            System.out.println(usage);
            System.exit(1);
        }

        String tableName = args[0];
        String key = args[1];
        String keyVal = args[2];
        String albumTitle = args[3];
        String albumTitleValue = args[4];
        String awards = args[5];
        String awardVal = args[6];
        String songTitle = args[7];
        String songTitleVal = args[8];

        Region region = Region.US_EAST_1;
        DynamoDbClient ddb = DynamoDbClient.builder()
                .region(region)
                .build();

        putItemInTable(ddb, tableName, key, keyVal, albumTitle, albumTitleValue,
awards, awardVal, songTitle,
                songTitleVal);
        System.out.println("Done!");
        ddb.close();
    }
```

```
    public static void putItemInTable(DynamoDbClient ddb,
            String tableName,
            String key,
            String keyVal,
            String albumTitle,
            String albumTitleValue,
            String awards,
            String awardVal,
            String songTitle,
            String songTitleVal) {

        HashMap<String, AttributeValue> itemValues = new HashMap<>();
        itemValues.put(key, AttributeValue.builder().s(keyVal).build());
        itemValues.put(songTitle,
 AttributeValue.builder().s(songTitleVal).build());
        itemValues.put(albumTitle,
 AttributeValue.builder().s(albumTitleValue).build());
        itemValues.put(awards, AttributeValue.builder().s(awardVal).build());

        PutItemRequest request = PutItemRequest.builder()
                .tableName(tableName)
                .item(itemValues)
                .build();

        try {
            PutItemResponse response = ddb.putItem(request);
            System.out.println(tableName + " was successfully updated. The
 request id is "
                    + response.responseMetadata().requestId());

        } catch (ResourceNotFoundException e) {
            System.err.format("Error: The Amazon DynamoDB table \"%s\" can't be
 found.\n", tableName);
            System.err.println("Be sure that it exists and that you've typed its
 name correctly!");
            System.exit(1);
        } catch (DynamoDbException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }
}
```

- For API details, see PutItem in *AWS SDK for Java 2.x API Reference*.

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

This example uses the document client to simplify working with items in DynamoDB. For API details see PutCommand.

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
import { PutCommand, DynamoDBDocumentClient } from "@aws-sdk/lib-dynamodb";

const client = new DynamoDBClient({});
const docClient = DynamoDBDocumentClient.from(client);

export const main = async () => {
  const command = new PutCommand({
    TableName: "HappyAnimals",
    Item: {
      CommonName: "Shiba Inu",
    },
  });

  const response = await docClient.send(command);
  console.log(response);
  return response;
};
```

- For API details, see PutItem in *AWS SDK for JavaScript API Reference*.

SDK for JavaScript (v2)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Put an item in a table.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create the DynamoDB service object
var ddb = new AWS.DynamoDB({ apiVersion: "2012-08-10" });

var params = {
  TableName: "CUSTOMER_LIST",
  Item: {
    CUSTOMER_ID: { N: "001" },
    CUSTOMER_NAME: { S: "Richard Roe" },
  },
};

// Call DynamoDB to add the item to the table
ddb.putItem(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data);
  }
});
```

Put an item in a table using the DynamoDB document client.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });
```

```
// Create DynamoDB document client
var docClient = new AWS.DynamoDB.DocumentClient({ apiVersion: "2012-08-10" });

var params = {
  TableName: "TABLE",
  Item: {
    HASHKEY: VALUE,
    ATTRIBUTE_1: "STRING_VALUE",
    ATTRIBUTE_2: VALUE_2,
  },
};

docClient.put(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data);
  }
});
```

- For more information, see AWS SDK for JavaScript Developer Guide.
- For API details, see PutItem in *AWS SDK for JavaScript API Reference*.

Kotlin

SDK for Kotlin

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
suspend fun putItemInTable(
    tableNameVal: String,
    key: String,
    keyVal: String,
    albumTitle: String,
    albumTitleValue: String,
```

```
    awards: String,
    awardVal: String,
    songTitle: String,
    songTitleVal: String,
) {
    val itemValues = mutableMapOf<String, AttributeValue>()

    // Add all content to the table.
    itemValues[key] = AttributeValue.S(keyVal)
    itemValues[songTitle] = AttributeValue.S(songTitleVal)
    itemValues[albumTitle] = AttributeValue.S(albumTitleValue)
    itemValues[awards] = AttributeValue.S(awardVal)

    val request =
        PutItemRequest {
            tableName = tableNameVal
            item = itemValues
        }

    DynamoDbClient.fromEnvironment { region = "us-east-1" }.use { ddb ->
        ddb.putItem(request)
        println(" A new item was placed into $tableNameVal.")
    }
}
```

- For API details, see PutItem in *AWS SDK for Kotlin API reference*.

PHP

SDK for PHP

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
        echo "What's the name of the last movie you watched?\n";
        while (empty($movieName)) {
            $movieName = testable_readline("Movie name: ");
        }
```

```
        echo "And what year was it released?\n";
        $movieYear = "year";
        while (!is_numeric($movieYear) || intval($movieYear) != $movieYear) {
            $movieYear = testable_readline("Year released: ");
        }

        $service->putItem([
            'Item' => [
                'year' => [
                    'N' => "$movieYear",
                ],
                'title' => [
                    'S' => $movieName,
                ],
            ],
            'TableName' => $tableName,
        ]);

    public function putItem(array $array)
    {
        $this->dynamoDbClient->putItem($array);
    }
```

- For API details, see PutItem in *AWS SDK for PHP API Reference*.

PowerShell

Tools for PowerShell V4

Example 1: Creates a new item, or replaces an existing item with a new item.

```
$item = @{
  SongTitle = 'Somewhere Down The Road'
  Artist = 'No One You Know'
        AlbumTitle = 'Somewhat Famous'
        Price = 1.94
        Genre = 'Country'
        CriticRating = 9.0
} | ConvertTo-DDBItem
Set-DDBItem -TableName 'Music' -Item $item
```

- For API details, see PutItem in *AWS Tools for PowerShell Cmdlet Reference (V4)*.

Tools for PowerShell V5

Example 1: Creates a new item, or replaces an existing item with a new item.

```
$item = @{
  SongTitle = 'Somewhere Down The Road'
  Artist = 'No One You Know'
        AlbumTitle = 'Somewhat Famous'
        Price = 1.94
        Genre = 'Country'
        CriticRating = 9.0
} | ConvertTo-DDBItem
Set-DDBItem -TableName 'Music' -Item $item
```

- For API details, see PutItem in *AWS Tools for PowerShell Cmdlet Reference (V5)*.

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
class Movies:
    """Encapsulates an Amazon DynamoDB table of movie data.

    Example data structure for a movie record in this table:
        {
            "year": 1999,
            "title": "For Love of the Game",
            "info": {
                "directors": ["Sam Raimi"],
                "release_date": "1999-09-15T00:00:00Z",
                "rating": 6.3,
                "plot": "A washed up pitcher flashes through his career.",
                "rank": 4987,
                "running_time_secs": 8220,
                "actors": [
```

```
                "Kevin Costner",
                "Kelly Preston",
                "John C. Reilly"
            ]
        }
    }
"""

def __init__(self, dyn_resource):
    """
    :param dyn_resource: A Boto3 DynamoDB resource.
    """
    self.dyn_resource = dyn_resource
    # The table variable is set during the scenario in the call to
    # 'exists' if the table exists. Otherwise, it is set by 'create_table'.
    self.table = None


def add_movie(self, title, year, plot, rating):
    """
    Adds a movie to the table.

    :param title: The title of the movie.
    :param year: The release year of the movie.
    :param plot: The plot summary of the movie.
    :param rating: The quality rating of the movie.
    """
    try:
        self.table.put_item(
            Item={
                "year": year,
                "title": title,
                "info": {"plot": plot, "rating": Decimal(str(rating))},
            }
        )
    except ClientError as err:
        logger.error(
            "Couldn't add movie %s to table %s. Here's why: %s: %s",
            title,
            self.table.name,
            err.response["Error"]["Code"],
            err.response["Error"]["Message"],
        )
        raise
```

- For API details, see PutItem in *AWS SDK for Python (Boto3) API Reference*.

Ruby

SDK for Ruby

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
class DynamoDBBasics
  attr_reader :dynamo_resource, :table

  def initialize(table_name)
    client = Aws::DynamoDB::Client.new(region: 'us-east-1')
    @dynamo_resource = Aws::DynamoDB::Resource.new(client: client)
    @table = @dynamo_resource.table(table_name)
  end

  # Adds a movie to the table.
  #
  # @param movie [Hash] The title, year, plot, and rating of the movie.
  def add_item(movie)
    @table.put_item(
      item: {
        'year' => movie[:year],
        'title' => movie[:title],
        'info' => { 'plot' => movie[:plot], 'rating' => movie[:rating] }
      }
    )
  rescue Aws::DynamoDB::Errors::ServiceError => e
    puts("Couldn't add movie #{title} to table #{@table.name}. Here's why:")
    puts("\t#{e.code}: #{e.message}")
    raise
  end
```

- For API details, see PutItem in *AWS SDK for Ruby API Reference*.

Rust

SDK for Rust

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
pub async fn add_item(client: &Client, item: Item, table: &String) ->
 Result<ItemOut, Error> {
    let user_av = AttributeValue::S(item.username);
    let type_av = AttributeValue::S(item.p_type);
    let age_av = AttributeValue::S(item.age);
    let first_av = AttributeValue::S(item.first);
    let last_av = AttributeValue::S(item.last);

    let request = client
        .put_item()
        .table_name(table)
        .item("username", user_av)
        .item("account_type", type_av)
        .item("age", age_av)
        .item("first_name", first_av)
        .item("last_name", last_av);

    println!("Executing request [{request:?}] to add item...");

    let resp = request.send().await?;

    let attributes = resp.attributes().unwrap();

    let username = attributes.get("username").cloned();
    let first_name = attributes.get("first_name").cloned();
    let last_name = attributes.get("last_name").cloned();
    let age = attributes.get("age").cloned();
    let p_type = attributes.get("p_type").cloned();

    println!(
```

```
        "Added user {:?}, {:?} {:?}, age {:?} as {:?} user",
        username, first_name, last_name, age, p_type
    );

    Ok(ItemOut {
        p_type,
        age,
        username,
        first_name,
        last_name,
    })
}
```

- For API details, see PutItem in *AWS SDK for Rust API reference*.

SAP ABAP

SDK for SAP ABAP

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    TRY.
        DATA(lo_resp) = lo_dyn->putitem(
          iv_tablename = iv_table_name
          it_item      = it_item ).
        MESSAGE '1 row inserted into DynamoDB Table' && iv_table_name TYPE 'I'.
      CATCH /aws1/cx_dyncondalcheckfaile00.
        MESSAGE 'A condition specified in the operation could not be evaluated.'
 TYPE 'E'.
      CATCH /aws1/cx_dynresourcenotfoundex.
        MESSAGE 'The table or index does not exist' TYPE 'E'.
      CATCH /aws1/cx_dyntransactconflictex.
        MESSAGE 'Another transaction is using the item' TYPE 'E'.
    ENDTRY.
```

- For API details, see PutItem in *AWS SDK for SAP ABAP API reference*.

Swift

SDK for Swift

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import AWSDynamoDB


    /// Add a movie specified as a `Movie` structure to the Amazon DynamoDB
    /// table.
    ///
    /// - Parameter movie: The `Movie` to add to the table.
    ///
    func add(movie: Movie) async throws {
        do {
            guard let client = self.ddbClient else {
                throw MoviesError.UninitializedClient
            }

            // Get a DynamoDB item containing the movie data.
            let item = try await movie.getAsItem()

            // Send the `PutItem` request to Amazon DynamoDB.

            let input = PutItemInput(
                item: item,
                tableName: self.tableName
            )
            _ = try await client.putItem(input: input)
        } catch {
            print("ERROR: add movie:", dump(error))
            throw error
        }
    }
```

```
    ///
    /// Return an array mapping attribute names to Amazon DynamoDB attribute
    /// values, representing the contents of the `Movie` record as a DynamoDB
    /// item.
    ///
    /// - Returns: The movie item as an array of type
    ///   `[Swift.String:DynamoDBClientTypes.AttributeValue]`.
    ///
    func getAsItem() async throws ->
[Swift.String:DynamoDBClientTypes.AttributeValue]  {
        // Build the item record, starting with the year and title, which are
        // always present.

        var item: [Swift.String:DynamoDBClientTypes.AttributeValue] = [
            "year": .n(String(self.year)),
            "title": .s(self.title)
        ]

        // Add the `info` field with the rating and/or plot if they're
        // available.

        var details: [Swift.String:DynamoDBClientTypes.AttributeValue] = [:]
        if (self.info.rating != nil || self.info.plot != nil) {
            if self.info.rating != nil {
                details["rating"] = .n(String(self.info.rating!))
            }
            if self.info.plot != nil {
                details["plot"] = .s(self.info.plot!)
            }
        }
        item["info"] = .m(details)

        return item
    }
```

- For API details, see PutItem in *AWS SDK for Swift API reference*.

For more DynamoDB examples, see Code examples for DynamoDB using AWS SDKs.

After writing data to your table, proceed to Step 3: Read data from a DynamoDB table.

Step 3: Read data from a DynamoDB table

In this step, you'll read back one of the items that you created in Step 2: Write data to a DynamoDB table. You can use the DynamoDB console or the AWS CLI to read an item from the `Music` table by specifying `Artist` and `SongTitle`.

For more information about read operations in DynamoDB, see Reading an item.

AWS Management Console

Follow these steps to read data from the `Music` table using the DynamoDB console.

1. Open the DynamoDB console at https://console.aws.amazon.com/dynamodb/.
2. In the left navigation pane, choose **Tables**.
3. On the **Tables** page, choose the **Music** table.
4. Choose **Explore table items**.
5. On the **Items returned** section, view the list of items stored in the table, sorted by `Artist` and `SongTitle`. The first item in the list is the one with the **Artist** named **Acme Band** and the **SongTitle PartiQL Rocks**.

AWS CLI

The following AWS CLI example reads an item from the `Music`. You can do this either through the DynamoDB API or PartiQL, a SQL-compatible query language for DynamoDB.

DynamoDB API

> **Note**
>
> The default behavior for DynamoDB is eventually consistent reads. The `consistent-read` parameter is used below to demonstrate strongly consistent reads.

Linux

```
aws dynamodb get-item --consistent-read \
    --table-name Music \
    --key '{ "Artist": {"S": "Acme Band"}, "SongTitle": {"S": "Happy Day"}}'
```

Windows CMD

```
aws dynamodb get-item --consistent-read ^
    --table-name Music ^
    --key "{\"Artist\": {\"S\": \"Acme Band\"}, \"SongTitle\": {\"S\": \"Happy Day
\"}}"
```

Using `get-item` returns the following sample result.

```
{
    "Item": {
        "AlbumTitle": {
            "S": "Songs About Life"
        },
        "Awards": {
            "S": "10"
        },
        "Artist": {
            "S": "Acme Band"
        },
        "SongTitle": {
            "S": "Happy Day"
        }
    }
}
```

PartiQL for DynamoDB

Linux

```
aws dynamodb execute-statement --statement "SELECT * FROM Music   \
WHERE Artist='Acme Band' AND SongTitle='Happy Day'"
```

Windows CMD

```
aws dynamodb execute-statement --statement "SELECT * FROM Music WHERE Artist='Acme
 Band' AND SongTitle='Happy Day'"
```

Using the PartiQL `Select` statement returns the following sample result.

```
{
```

```
    "Items": [
        {
            "AlbumTitle": {
                "S": "Songs About Life"
            },
            "Awards": {
                "S": "10"
            },
            "Artist": {
                "S": "Acme Band"
            },
            "SongTitle": {
                "S": "Happy Day"
            }
        }
    ]
}
```

For more information about reading data with PartiQL, see PartiQL select statements.

AWS SDK

The following code examples show how to read an item from a DynamoDB table using an AWS SDK.

.NET

SDK for .NET (v4)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    /// <summary>
    /// Gets information about an existing movie from the table.
    /// </summary>
    /// <param name="newMovie">A Movie object containing information about
    /// the movie to retrieve.</param>
```

```
    /// <param name="tableName">The name of the table containing the movie.</
param>
    /// <returns>A Dictionary object containing information about the item
    /// retrieved.</returns>
    public async Task<Dictionary<string, AttributeValue>> GetItemAsync(Movie
 newMovie, string tableName)
    {
        try
        {
            var key = new Dictionary<string, AttributeValue>
            {
                ["title"] = new AttributeValue { S = newMovie.Title },
                ["year"] = new AttributeValue { N = newMovie.Year.ToString() },
            };

            var request = new GetItemRequest
            {
                Key = key,
                TableName = tableName,
            };

            var response = await _amazonDynamoDB.GetItemAsync(request);
            return response.Item;
        }
        catch (ResourceNotFoundException ex)
        {
            Console.WriteLine($"Table {tableName} was not found. {ex.Message}");
            return new Dictionary<string, AttributeValue>();
        }
        catch (AmazonDynamoDBException ex)
        {
            Console.WriteLine($"An Amazon DynamoDB error occurred while getting
 item. {ex.Message}");
            throw;
        }
        catch (Exception ex)
        {
            Console.WriteLine($"An error occurred while getting item.
 {ex.Message}");
            throw;
        }
    }
```

- For API details, see GetItem in *AWS SDK for .NET API Reference*.

Bash

AWS CLI with Bash script

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
###############################################################################
# function dynamodb_get_item
#
# This function gets an item from a DynamoDB table.
#
# Parameters:
#       -n table_name  -- The name of the table.
#       -k keys  -- Path to json file containing the keys that identify the item
 to get.
#       [-q query]  -- Optional JMESPath query expression.
#
#  Returns:
#       The item as text output.
#  And:
#       0 - If successful.
#       1 - If it fails.
###############################################################################
function dynamodb_get_item() {
  local table_name keys query response
  local option OPTARG # Required to use getopts command in a function.

  # ######################################
  # Function usage explanation
  ######################################
  function usage() {
    echo "function dynamodb_get_item"
    echo "Get an item from a DynamoDB table."
    echo " -n table_name  -- The name of the table."
```

```
    echo " -k keys  -- Path to json file containing the keys that identify the
item to get."
    echo " [-q query]  -- Optional JMESPath query expression."
    echo ""
 }
 query=""
 while getopts "n:k:q:h" option; do
    case "${option}" in
      n) table_name="${OPTARG}" ;;
      k) keys="${OPTARG}" ;;
      q) query="${OPTARG}" ;;
      h)
        usage
        return 0
        ;;
      \?)
        echo "Invalid parameter"
        usage
        return 1
        ;;
    esac
 done
 export OPTIND=1

 if [[ -z "$table_name" ]]; then
    errecho "ERROR: You must provide a table name with the -n parameter."
    usage
    return 1
 fi

 if [[ -z "$keys" ]]; then
    errecho "ERROR: You must provide a keys json file path the -k parameter."
    usage
    return 1
 fi

 if [[ -n "$query" ]]; then
    response=$(aws dynamodb get-item \
      --table-name "$table_name" \
      --key file://"$keys" \
      --output text \
      --query "$query")
 else
    response=$(
```

```
      aws dynamodb get-item \
        --table-name "$table_name" \
        --key file://"$keys" \
        --output text
    )
  fi

  local error_code=${?}

  if [[ $error_code -ne 0 ]]; then
    aws_cli_error_log $error_code
    errecho "ERROR: AWS reports get-item operation failed.$response"
    return 1
  fi

  if [[ -n "$query" ]]; then
    echo "$response" | sed "/^\t/s/\t//1" # Remove initial tab that the JMSEPath
 query inserts on some strings.
  else
    echo "$response"
  fi

  return 0
}
```

The utility functions used in this example.

```
###############################################################################
# function errecho
#
# This function outputs everything sent to it to STDERR (standard error output).
###############################################################################
function errecho() {
  printf "%s\n" "$*" 1>&2
}

###############################################################################
# function aws_cli_error_log()
#
# This function is used to log the error messages from the AWS CLI.
#
```

```
# See https://docs.aws.amazon.com/cli/latest/topic/return-codes.html#cli-aws-
help-return-codes.
#
# The function expects the following argument:
#         $1 - The error code returned by the AWS CLI.
#
#  Returns:
#          0: - Success.
#
###############################################################################
function aws_cli_error_log() {
  local err_code=$1
  errecho "Error code : $err_code"
  if [ "$err_code" == 1 ]; then
    errecho "  One or more S3 transfers failed."
  elif [ "$err_code" == 2 ]; then
    errecho "  Command line failed to parse."
  elif [ "$err_code" == 130 ]; then
    errecho "  Process received SIGINT."
  elif [ "$err_code" == 252 ]; then
    errecho "  Command syntax invalid."
  elif [ "$err_code" == 253 ]; then
    errecho "  The system environment or configuration was invalid."
  elif [ "$err_code" == 254 ]; then
    errecho "  The service returned an error."
  elif [ "$err_code" == 255 ]; then
    errecho "  255 is a catch-all error."
  fi

  return 0
}
```

- For API details, see GetItem in *AWS CLI Command Reference*.

C++

SDK for C++

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
//! Get an item from an Amazon DynamoDB table.
/*!
  \sa getItem()
  \param tableName: The table name.
  \param partitionKey: The partition key.
  \param partitionValue: The value for the partition key.
  \param clientConfiguration: AWS client configuration.
  \return bool: Function succeeded.
 */

bool AwsDoc::DynamoDB::getItem(const Aws::String &tableName,
                               const Aws::String &partitionKey,
                               const Aws::String &partitionValue,
                               const Aws::Client::ClientConfiguration
 &clientConfiguration) {
    Aws::DynamoDB::DynamoDBClient dynamoClient(clientConfiguration);
    Aws::DynamoDB::Model::GetItemRequest request;

    // Set up the request.
    request.SetTableName(tableName);
    request.AddKey(partitionKey,
                   Aws::DynamoDB::Model::AttributeValue().SetS(partitionValue));

    // Retrieve the item's fields and values.
    const Aws::DynamoDB::Model::GetItemOutcome &outcome =
 dynamoClient.GetItem(request);
    if (outcome.IsSuccess()) {
        // Reference the retrieved fields/values.
        const Aws::Map<Aws::String, Aws::DynamoDB::Model::AttributeValue> &item =
 outcome.GetResult().GetItem();
        if (!item.empty()) {
            // Output each retrieved field and its value.
```

```
            for (const auto &i: item)
                std::cout << "Values: " << i.first << ": " << i.second.GetS()
                          << std::endl;
        }
        else {
            std::cout << "No item found with the key " << partitionKey <<
 std::endl;
        }
    }
    else {
        std::cout << "Failed to get item: " << outcome.GetError().GetMessage();
    }

    return outcome.IsSuccess();
}
```

- For API details, see GetItem in *AWS SDK for C++ API Reference*.

CLI

AWS CLI

Example 1: To read an item in a table

The following `get-item` example retrieves an item from the `MusicCollection` table. The table has a hash-and-range primary key (`Artist` and `SongTitle`), so you must specify both of these attributes. The command also requests information about the read capacity consumed by the operation.

```
aws dynamodb get-item \
    --table-name MusicCollection \
    --key file://key.json \
    --return-consumed-capacity TOTAL
```

Contents of `key.json`:

```
{
    "Artist": {"S": "Acme Band"},
    "SongTitle": {"S": "Happy Day"}
}
```

Output:

```
{
    "Item": {
        "AlbumTitle": {
            "S": "Songs About Life"
        },
        "SongTitle": {
            "S": "Happy Day"
        },
        "Artist": {
            "S": "Acme Band"
        }
    },
    "ConsumedCapacity": {
        "TableName": "MusicCollection",
        "CapacityUnits": 0.5
    }
}
```

For more information, see Reading an Item in the *Amazon DynamoDB Developer Guide*.

Example 2: To read an item using a consistent read

The following example retrieves an item from the `MusicCollection` table using strongly consistent reads.

```
aws dynamodb get-item \
    --table-name MusicCollection \
    --key file://key.json \
    --consistent-read \
    --return-consumed-capacity TOTAL
```

Contents of `key.json`:

```
{
    "Artist": {"S": "Acme Band"},
    "SongTitle": {"S": "Happy Day"}
}
```

Output:

```
{
    "Item": {
        "AlbumTitle": {
            "S": "Songs About Life"
        },
        "SongTitle": {
            "S": "Happy Day"
        },
        "Artist": {
            "S": "Acme Band"
        }
    },
    "ConsumedCapacity": {
        "TableName": "MusicCollection",
        "CapacityUnits": 1.0
    }
}
```

For more information, see Reading an Item in the *Amazon DynamoDB Developer Guide*.

Example 3: To retrieve specific attributes of an item

The following example uses a projection expression to retrieve only three attributes of the desired item.

```
aws dynamodb get-item \
    --table-name ProductCatalog \
    --key '{"Id": {"N": "102"}}' \
    --projection-expression "#T, #C, #P" \
    --expression-attribute-names file://names.json
```

Contents of `names.json`:

```
{
    "#T": "Title",
    "#C": "ProductCategory",
    "#P": "Price"
}
```

Output:

```
{
```

```
    "Item": {
        "Price": {
            "N": "20"
        },
        "Title": {
            "S": "Book 102 Title"
        },
        "ProductCategory": {
            "S": "Book"
        }
    }
}
```

For more information, see Reading an Item in the *Amazon DynamoDB Developer Guide*.

- For API details, see GetItem in *AWS CLI Command Reference*.

Go

SDK for Go V2

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import (
 "context"
 "errors"
 "log"
 "time"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/attributevalue"
 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/expression"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb/types"
)
```

```
// TableBasics encapsulates the Amazon DynamoDB service actions used in the
 examples.
// It contains a DynamoDB service client that is used to act on the specified
 table.
type TableBasics struct {
 DynamoDbClient *dynamodb.Client
 TableName      string
}



// GetMovie gets movie data from the DynamoDB table by using the primary
 composite key
// made of title and year.
func (basics TableBasics) GetMovie(ctx context.Context, title string, year int)
 (Movie, error) {
 movie := Movie{Title: title, Year: year}
 response, err := basics.DynamoDbClient.GetItem(ctx, &dynamodb.GetItemInput{
  Key: movie.GetKey(), TableName: aws.String(basics.TableName),
 })
 if err != nil {
  log.Printf("Couldn't get info about %v. Here's why: %v\n", title, err)
 } else {
  err = attributevalue.UnmarshalMap(response.Item, &movie)
  if err != nil {
   log.Printf("Couldn't unmarshal response. Here's why: %v\n", err)
  }
 }
 return movie, err
}
```

Define a Movie struct that is used in this example.

```
import (
 "archive/zip"
 "bytes"
 "encoding/json"
 "fmt"
 "io"
 "log"
```

```
 "net/http"

 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/attributevalue"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb/types"
)

// Movie encapsulates data about a movie. Title and Year are the composite
 primary key
// of the movie in Amazon DynamoDB. Title is the sort key, Year is the partition
 key,
// and Info is additional data.
type Movie struct {
 Title string                 `dynamodbav:"title"`
 Year  int                    `dynamodbav:"year"`
 Info  map[string]interface{} `dynamodbav:"info"`
}

// GetKey returns the composite primary key of the movie in a format that can be
// sent to DynamoDB.
func (movie Movie) GetKey() map[string]types.AttributeValue {
 title, err := attributevalue.Marshal(movie.Title)
 if err != nil {
  panic(err)
 }
 year, err := attributevalue.Marshal(movie.Year)
 if err != nil {
  panic(err)
 }
 return map[string]types.AttributeValue{"title": title, "year": year}
}

// String returns the title, year, rating, and plot of a movie, formatted for the
 example.
func (movie Movie) String() string {
 return fmt.Sprintf("%v\n\tReleased: %v\n\tRating: %v\n\tPlot: %v\n",
  movie.Title, movie.Year, movie.Info["rating"], movie.Info["plot"])
}
```

- For API details, see GetItem in *AWS SDK for Go API Reference*.

Java

SDK for Java 2.x

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Gets an item from a table by using the DynamoDbClient.

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.GetItemRequest;
import java.util.HashMap;
import java.util.Map;
import java.util.Set;

/**
 * Before running this Java V2 code example, set up your development
 * environment, including your credentials.
 *
 * For more information, see the following documentation topic:
 *
 * https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-
started.html
 *
 * To get an item from an Amazon DynamoDB table using the AWS SDK for Java V2,
 * its better practice to use the
 * Enhanced Client, see the EnhancedGetItem example.
 */
public class GetItem {
    public static void main(String[] args) {
        final String usage = """

                Usage:
                    <tableName> <key> <keyVal>

                Where:
```

```
                    tableName - The Amazon DynamoDB table from which an item is
retrieved (for example, Music3).\s
                    key - The key used in the Amazon DynamoDB table (for example,
Artist).\s
                    keyval - The key value that represents the item to get (for
example, Famous Band).
                """;

       if (args.length != 3) {
           System.out.println(usage);
           System.exit(1);
       }

       String tableName = args[0];
       String key = args[1];
       String keyVal = args[2];
       System.out.format("Retrieving item \"%s\" from \"%s\"\n", keyVal,
tableName);
       Region region = Region.US_EAST_1;
       DynamoDbClient ddb = DynamoDbClient.builder()
               .region(region)
               .build();

       getDynamoDBItem(ddb, tableName, key, keyVal);
       ddb.close();
   }

   public static void getDynamoDBItem(DynamoDbClient ddb, String tableName,
String key, String keyVal) {
       HashMap<String, AttributeValue> keyToGet = new HashMap<>();
       keyToGet.put(key, AttributeValue.builder()
               .s(keyVal)
               .build());

       GetItemRequest request = GetItemRequest.builder()
               .key(keyToGet)
               .tableName(tableName)
               .build();

       try {
           // If there is no matching item, GetItem does not return any data.
           Map<String, AttributeValue> returnedItem =
ddb.getItem(request).item();
           if (returnedItem.isEmpty())
```

```
                System.out.format("No item found with the key %s!\n", key);
            else {
                Set<String> keys = returnedItem.keySet();
                System.out.println("Amazon DynamoDB table attributes: \n");
                for (String key1 : keys) {
                    System.out.format("%s: %s\n", key1,
 returnedItem.get(key1).toString());
                }
            }

        } catch (DynamoDbException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }
}
```

- For API details, see GetItem in *AWS SDK for Java 2.x API Reference*.

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

This example uses the document client to simplify working with items in DynamoDB. For API details see GetCommand.

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
import { DynamoDBDocumentClient, GetCommand } from "@aws-sdk/lib-dynamodb";

const client = new DynamoDBClient({});
const docClient = DynamoDBDocumentClient.from(client);

export const main = async () => {
  const command = new GetCommand({
    TableName: "AngryAnimals",
```

```
    Key: {
      CommonName: "Shoebill",
    },
  });

  const response = await docClient.send(command);
  console.log(response);
  return response;
};
```

- For API details, see GetItem in *AWS SDK for JavaScript API Reference*.

SDK for JavaScript (v2)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Get an item from a table.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create the DynamoDB service object
var ddb = new AWS.DynamoDB({ apiVersion: "2012-08-10" });

var params = {
  TableName: "TABLE",
  Key: {
    KEY_NAME: { N: "001" },
  },
  ProjectionExpression: "ATTRIBUTE_NAME",
};

// Call DynamoDB to read the item from the table
ddb.getItem(params, function (err, data) {
  if (err) {
    console.log("Error", err);
```

```
  } else {
    console.log("Success", data.Item);
  }
});
```

Get an item from a table using the DynamoDB document client.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create DynamoDB document client
var docClient = new AWS.DynamoDB.DocumentClient({ apiVersion: "2012-08-10" });

var params = {
  TableName: "EPISODES_TABLE",
  Key: { KEY_NAME: VALUE },
};

docClient.get(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data.Item);
  }
});
```

- For more information, see AWS SDK for JavaScript Developer Guide.
- For API details, see GetItem in *AWS SDK for JavaScript API Reference*.

Kotlin

SDK for Kotlin

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
suspend fun getSpecificItem(
    tableNameVal: String,
    keyName: String,
    keyVal: String,
) {
    val keyToGet = mutableMapOf<String, AttributeValue>()
    keyToGet[keyName] = AttributeValue.S(keyVal)

    val request =
        GetItemRequest {
            key = keyToGet
            tableName = tableNameVal
        }

    DynamoDbClient.fromEnvironment { region = "us-east-1" }.use { ddb ->
        val returnedItem = ddb.getItem(request)
        val numbersMap = returnedItem.item
        numbersMap?.forEach { key1 ->
            println(key1.key)
            println(key1.value)
        }
    }
}
```

- For API details, see GetItem in *AWS SDK for Kotlin API reference*.

PHP

SDK for PHP

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
        $movie = $service->getItemByKey($tableName, $key);
        echo "\nThe movie {$movie['Item']['title']['S']} was released in
{$movie['Item']['year']['N']}.\n";
```

```
    public function getItemByKey(string $tableName, array $key)
    {
        return $this->dynamoDbClient->getItem([
            'Key' => $key['Item'],
            'TableName' => $tableName,
        ]);
    }
```

- For API details, see GetItem in *AWS SDK for PHP API Reference*.

PowerShell

Tools for PowerShell V4

Example 1: Returns the DynamoDB item with the partition key SongTitle and the sort key Artist.

```
$key = @{
  SongTitle = 'Somewhere Down The Road'
  Artist = 'No One You Know'
} | ConvertTo-DDBItem

Get-DDBItem -TableName 'Music' -Key $key | ConvertFrom-DDBItem
```

Output:

```
Name                           Value
----                           -----
Genre                          Country
SongTitle                      Somewhere Down The Road
Price                          1.94
Artist                         No One You Know
CriticRating                   9
AlbumTitle                     Somewhat Famous
```

- For API details, see GetItem in *AWS Tools for PowerShell Cmdlet Reference (V4)*.

Tools for PowerShell V5

Example 1: Returns the DynamoDB item with the partition key SongTitle and the sort key Artist.

```
$key = @{
  SongTitle = 'Somewhere Down The Road'
  Artist = 'No One You Know'
} | ConvertTo-DDBItem

Get-DDBItem -TableName 'Music' -Key $key | ConvertFrom-DDBItem
```

Output:

```
Name                           Value
----                           -----
Genre                          Country
SongTitle                      Somewhere Down The Road
Price                          1.94
Artist                         No One You Know
CriticRating                   9
AlbumTitle                     Somewhat Famous
```

- For API details, see GetItem in *AWS Tools for PowerShell Cmdlet Reference (V5)*.

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
class Movies:
    """Encapsulates an Amazon DynamoDB table of movie data.

    Example data structure for a movie record in this table:
        {
            "year": 1999,
```

```
            "title": "For Love of the Game",
            "info": {
                "directors": ["Sam Raimi"],
                "release_date": "1999-09-15T00:00:00Z",
                "rating": 6.3,
                "plot": "A washed up pitcher flashes through his career.",
                "rank": 4987,
                "running_time_secs": 8220,
                "actors": [
                    "Kevin Costner",
                    "Kelly Preston",
                    "John C. Reilly"
                ]
            }
        }
    """

    def __init__(self, dyn_resource):
        """
        :param dyn_resource: A Boto3 DynamoDB resource.
        """
        self.dyn_resource = dyn_resource
        # The table variable is set during the scenario in the call to
        # 'exists' if the table exists. Otherwise, it is set by 'create_table'.
        self.table = None


    def get_movie(self, title, year):
        """
        Gets movie data from the table for a specific movie.

        :param title: The title of the movie.
        :param year: The release year of the movie.
        :return: The data about the requested movie.
        """
        try:
            response = self.table.get_item(Key={"year": year, "title": title})
        except ClientError as err:
            logger.error(
                "Couldn't get movie %s from table %s. Here's why: %s: %s",
                title,
                self.table.name,
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
```

```
            )
            raise
        else:
            return response["Item"]
```

- For API details, see GetItem in *AWS SDK for Python (Boto3) API Reference*.

Ruby

SDK for Ruby

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
class DynamoDBBasics
  attr_reader :dynamo_resource, :table

  def initialize(table_name)
    client = Aws::DynamoDB::Client.new(region: 'us-east-1')
    @dynamo_resource = Aws::DynamoDB::Resource.new(client: client)
    @table = @dynamo_resource.table(table_name)
  end

  # Gets movie data from the table for a specific movie.
  #
  # @param title [String] The title of the movie.
  # @param year [Integer] The release year of the movie.
  # @return [Hash] The data about the requested movie.
  def get_item(title, year)
    @table.get_item(key: { 'year' => year, 'title' => title })
  rescue Aws::DynamoDB::Errors::ServiceError => e
    puts("Couldn't get movie #{title} (#{year}) from table #{@table.name}:\n")
    puts("\t#{e.code}: #{e.message}")
    raise
  end
```

- For API details, see GetItem in *AWS SDK for Ruby API Reference*.

SAP ABAP

SDK for SAP ABAP

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
TRY.
    oo_item = lo_dyn->getitem(
      iv_tablename               = iv_table_name
      it_key                     = it_key ).
    DATA(lt_attr) = oo_item->get_item( ).
    DATA(lo_title) = lt_attr[ key = 'title' ]-value.
    DATA(lo_year) = lt_attr[ key = 'year' ]-value.
    DATA(lo_rating) = lt_attr[ key = 'rating' ]-value.
    MESSAGE 'Movie name is: ' && lo_title->get_s( )
      && 'Movie year is: ' && lo_year->get_n( )
      && 'Moving rating is: ' && lo_rating->get_n( ) TYPE 'I'.
  CATCH /aws1/cx_dynresourcenotfoundex.
    MESSAGE 'The table or index does not exist' TYPE 'E'.
ENDTRY.
```

- For API details, see GetItem in *AWS SDK for SAP ABAP API reference*.

Swift

SDK for Swift

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import AWSDynamoDB


    /// Return a `Movie` record describing the specified movie from the Amazon
    /// DynamoDB table.
    ///
    /// - Parameters:
    ///   - title: The movie's title (`String`).
    ///   - year: The movie's release year (`Int`).
    ///
    /// - Throws: `MoviesError.ItemNotFound` if the movie isn't in the table.
    ///
    /// - Returns: A `Movie` record with the movie's details.
    func get(title: String, year: Int) async throws -> Movie {
        do {
            guard let client = self.ddbClient else {
                throw MoviesError.UninitializedClient
            }

            let input = GetItemInput(
                key: [
                    "year": .n(String(year)),
                    "title": .s(title)
                ],
                tableName: self.tableName
            )
            let output = try await client.getItem(input: input)
            guard let item = output.item else {
                throw MoviesError.ItemNotFound
            }

            let movie = try Movie(withItem: item)
            return movie
        } catch {
            print("ERROR: get:", dump(error))
            throw error
        }
    }
```

- For API details, see GetItem in *AWS SDK for Swift API reference*.

For more DynamoDB examples, see Code examples for DynamoDB using AWS SDKs.

To update the data in your table, proceed to Step 4: Update data in a DynamoDB table.

Step 4: Update data in a DynamoDB table

In this step, you update an item that you created in Step 2: Write data to a DynamoDB table. You can use the DynamoDB console or the AWS CLI to update the `AlbumTitle` of an item in the `Music` table by specifying `Artist`, `SongTitle`, and the updated `AlbumTitle`.

For more information about write operations, see Writing an item.

AWS Management Console

You can use the DynamoDB console to update data in the `Music` table.

1. Open the DynamoDB console at https://console.aws.amazon.com/dynamodb/.
2. In the left navigation pane, choose **Tables**.
3. Choose the **Music** table from the table list.
4. Choose **Explore table items**.
5. In **Items returned**, for the item row with **Acme Band Artist** and **Happy Day SongTitle**, do the following:

    a. Place your cursor on the **AlbumTitle** named **Songs About Life**.

    b. Choose the Edit icon.

    c. In the **Edit String** popup window, enter **`Songs of Twilight`**.

    d. Choose **Save**.

> **Tip**
>
> Alternatively, to update an item, do the following in the **Items returned** section:
>
> 1. Choose the item row with **Artist** named **Acme Band** and **SongTitle** named **Happy Day**.
> 2. From the **Actions** dropdown list, choose **Edit item**.
> 3. For enter **AlbumTitle**, enter **`Songs of Twilight`**.

4. Choose **Save and close**.

AWS CLI

The following AWS CLI example updates an item in the `Music` table. You can do this either through the DynamoDB API or PartiQL, a SQL-compatible query language for DynamoDB.

DynamoDB API

Linux

```
aws dynamodb update-item \
    --table-name Music \
    --key '{ "Artist": {"S": "Acme Band"}, "SongTitle": {"S": "Happy Day"}}' \
    --update-expression "SET AlbumTitle = :newval" \
    --expression-attribute-values '{":newval":{"S":"Updated Album Title"}}' \
    --return-values ALL_NEW
```

Windows CMD

```
aws dynamodb update-item ^
    --table-name Music ^
    --key "{\"Artist\": {\"S\": \"Acme Band\"}, \"SongTitle\": {\"S\": \"Happy Day
\"}}" ^
    --update-expression "SET AlbumTitle = :newval" ^
    --expression-attribute-values "{\":newval\":{\"S\":\"Updated Album Title\"}}" ^
    --return-values ALL_NEW
```

Using `update-item` returns the following sample result because `return-values ALL_NEW` was specified.

```
{
    "Attributes": {
        "AlbumTitle": {
            "S": "Updated Album Title"
        },
        "Awards": {
            "S": "10"
        },
        "Artist": {
```

```
            "S": "Acme Band"
        },
        "SongTitle": {
            "S": "Happy Day"
        }
    }
}
```

PartiQL for DynamoDB

Linux

```
aws dynamodb execute-statement --statement "UPDATE Music  \
    SET AlbumTitle='Updated Album Title'  \
    WHERE Artist='Acme Band' AND SongTitle='Happy Day' \
    RETURNING ALL NEW *"
```

Windows CMD

```
aws dynamodb execute-statement --statement "UPDATE Music SET AlbumTitle='Updated
 Album Title' WHERE Artist='Acme Band' AND SongTitle='Happy Day' RETURNING ALL NEW
 *"
```

Using the `Update` statement returns the following sample result because `RETURNING ALL NEW *` was specified.

```
{
    "Items": [
        {
            "AlbumTitle": {
                "S": "Updated Album Title"
            },
            "Awards": {
                "S": "10"
            },
            "Artist": {
                "S": "Acme Band"
            },
            "SongTitle": {
                "S": "Happy Day"
            }
        }
```

```
    ]
}
```

For more information about updating data with PartiQL, see PartiQL update statements.

AWS SDK

The following code examples show how to update an item in a DynamoDB table using an AWS SDK.

.NET

SDK for .NET (v4)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    /// <summary>
    /// Updates an existing item in the movies table.
    /// </summary>
    /// <param name="newMovie">A Movie object containing information for
    /// the movie to update.</param>
    /// <param name="newInfo">A MovieInfo object that contains the
    /// information that will be changed.</param>
    /// <param name="tableName">The name of the table that contains the movie.</
param>
    /// <returns>A Boolean value that indicates the success of the operation.</
returns>
    public async Task<bool> UpdateItemAsync(
        Movie newMovie,
        MovieInfo newInfo,
        string tableName)
    {
        try
        {
            var key = new Dictionary<string, AttributeValue>
            {
```

```
                ["title"] = new AttributeValue { S = newMovie.Title },
                ["year"] = new AttributeValue { N = newMovie.Year.ToString() },
            };
            var updates = new Dictionary<string, AttributeValueUpdate>
            {
                ["info.plot"] = new AttributeValueUpdate
                {
                    Action = AttributeAction.PUT,
                    Value = new AttributeValue { S = newInfo.Plot },
                },

                ["info.rating"] = new AttributeValueUpdate
                {
                    Action = AttributeAction.PUT,
                    Value = new AttributeValue { N = newInfo.Rank.ToString() },
                },
            };

            var request = new UpdateItemRequest
            {
                AttributeUpdates = updates,
                Key = key,
                TableName = tableName,
            };

            await _amazonDynamoDB.UpdateItemAsync(request);
            return true;
        }
        catch (ResourceNotFoundException ex)
        {
            Console.WriteLine($"Table {tableName} or item was not found.
{ex.Message}");
            return false;
        }
        catch (AmazonDynamoDBException ex)
        {
            Console.WriteLine($"An Amazon DynamoDB error occurred while updating
item. {ex.Message}");
            throw;
        }
        catch (Exception ex)
        {
            Console.WriteLine($"An error occurred while updating item.
{ex.Message}");
```

```
            throw;
        }
    }
```

- For API details, see UpdateItem in *AWS SDK for .NET API Reference*.

Bash

AWS CLI with Bash script

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
###############################################################################
# function dynamodb_update_item
#
# This function updates an item in a DynamoDB table.
#
#
# Parameters:
#       -n table_name  -- The name of the table.
#       -k keys  -- Path to json file containing the keys that identify the item
 to update.
#       -e update expression  -- An expression that defines one or more
 attributes to be updated.
#       -v values  -- Path to json file containing the update values.
#
#  Returns:
#       0 - If successful.
#       1 - If it fails.
###############################################################################
function dynamodb_update_item() {
  local table_name keys update_expression values response
  local option OPTARG # Required to use getopts command in a function.

  ######################################
  # Function usage explanation
```

```
 ######################################
 function usage() {
   echo "function dynamodb_update_item"
   echo "Update an item in a DynamoDB table."
   echo " -n table_name  -- The name of the table."
   echo " -k keys  -- Path to json file containing the keys that identify the
item to update."
   echo " -e update expression  -- An expression that defines one or more
attributes to be updated."
   echo " -v values  -- Path to json file containing the update values."
   echo ""
 }

 while getopts "n:k:e:v:h" option; do
   case "${option}" in
     n) table_name="${OPTARG}" ;;
     k) keys="${OPTARG}" ;;
     e) update_expression="${OPTARG}" ;;
     v) values="${OPTARG}" ;;
     h)
       usage
       return 0
       ;;
     \?)
       echo "Invalid parameter"
       usage
       return 1
       ;;
   esac
 done
 export OPTIND=1

 if [[ -z "$table_name" ]]; then
   errecho "ERROR: You must provide a table name with the -n parameter."
   usage
   return 1
 fi

 if [[ -z "$keys" ]]; then
   errecho "ERROR: You must provide a keys json file path the -k parameter."
   usage
   return 1
 fi
 if [[ -z "$update_expression" ]]; then
```

```
    errecho "ERROR: You must provide an update expression with the -e parameter."
    usage
    return 1
  fi

  if [[ -z "$values" ]]; then
    errecho "ERROR: You must provide a values json file path the -v parameter."
    usage
    return 1
  fi

  iecho "Parameters:\n"
  iecho "    table_name:   $table_name"
  iecho "    keys:   $keys"
  iecho "    update_expression:   $update_expression"
  iecho "    values:   $values"

  response=$(aws dynamodb update-item \
    --table-name "$table_name" \
    --key file://"$keys" \
    --update-expression "$update_expression" \
    --expression-attribute-values file://"$values")

  local error_code=${?}

  if [[ $error_code -ne 0 ]]; then
    aws_cli_error_log $error_code
    errecho "ERROR: AWS reports update-item operation failed.$response"
    return 1
  fi

  return 0

}
```

The utility functions used in this example.

```
###############################################################################
# function iecho
#
# This function enables the script to display the specified text only if
# the global variable $VERBOSE is set to true.
```

```
#############################################################################
function iecho() {
  if [[ $VERBOSE == true ]]; then
    echo "$@"
  fi
}

#############################################################################
# function errecho
#
# This function outputs everything sent to it to STDERR (standard error output).
#############################################################################
function errecho() {
  printf "%s\n" "$*" 1>&2
}

#############################################################################
# function aws_cli_error_log()
#
# This function is used to log the error messages from the AWS CLI.
#
# See https://docs.aws.amazon.com/cli/latest/topic/return-codes.html#cli-aws-
help-return-codes.
#
# The function expects the following argument:
#         $1 - The error code returned by the AWS CLI.
#
#  Returns:
#          0: - Success.
#
#############################################################################
function aws_cli_error_log() {
  local err_code=$1
  errecho "Error code : $err_code"
  if [ "$err_code" == 1 ]; then
    errecho "  One or more S3 transfers failed."
  elif [ "$err_code" == 2 ]; then
    errecho "  Command line failed to parse."
  elif [ "$err_code" == 130 ]; then
    errecho "  Process received SIGINT."
  elif [ "$err_code" == 252 ]; then
    errecho "  Command syntax invalid."
  elif [ "$err_code" == 253 ]; then
    errecho "  The system environment or configuration was invalid."
```

```
  elif [ "$err_code" == 254 ]; then
    errecho "  The service returned an error."
  elif [ "$err_code" == 255 ]; then
    errecho "  255 is a catch-all error."
  fi

  return 0
}
```

- For API details, see UpdateItem in *AWS CLI Command Reference*.

C++

SDK for C++

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
//! Update an Amazon DynamoDB table item.
/*!
  \sa updateItem()
  \param tableName: The table name.
  \param partitionKey: The partition key.
  \param partitionValue: The value for the partition key.
  \param attributeKey: The key for the attribute to be updated.
  \param attributeValue: The value for the attribute to be updated.
  \param clientConfiguration: AWS client configuration.
  \return bool: Function succeeded.
  */

/*
 *  The example code only sets/updates an attribute value. It processes
 *  the attribute value as a string, even if the value could be interpreted
 *  as a number. Also, the example code does not remove an existing attribute
 *  from the key value.
 */

bool AwsDoc::DynamoDB::updateItem(const Aws::String &tableName,
```

```
                                    const Aws::String &partitionKey,
                                    const Aws::String &partitionValue,
                                    const Aws::String &attributeKey,
                                    const Aws::String &attributeValue,
                                    const Aws::Client::ClientConfiguration
&clientConfiguration) {
   Aws::DynamoDB::DynamoDBClient dynamoClient(clientConfiguration);

   // *** Define UpdateItem request arguments.
   // Define TableName argument.
   Aws::DynamoDB::Model::UpdateItemRequest request;
   request.SetTableName(tableName);

   // Define KeyName argument.
   Aws::DynamoDB::Model::AttributeValue attribValue;
   attribValue.SetS(partitionValue);
   request.AddKey(partitionKey, attribValue);

   // Construct the SET update expression argument.
   Aws::String update_expression("SET #a = :valueA");
   request.SetUpdateExpression(update_expression);

   // Construct attribute name argument.
   Aws::Map<Aws::String, Aws::String> expressionAttributeNames;
   expressionAttributeNames["#a"] = attributeKey;
   request.SetExpressionAttributeNames(expressionAttributeNames);

   // Construct attribute value argument.
   Aws::DynamoDB::Model::AttributeValue attributeUpdatedValue;
   attributeUpdatedValue.SetS(attributeValue);
   Aws::Map<Aws::String, Aws::DynamoDB::Model::AttributeValue>
expressionAttributeValues;
   expressionAttributeValues[":valueA"] = attributeUpdatedValue;
   request.SetExpressionAttributeValues(expressionAttributeValues);

   // Update the item.
   const Aws::DynamoDB::Model::UpdateItemOutcome &outcome =
dynamoClient.UpdateItem(
           request);
   if (outcome.IsSuccess()) {
       std::cout << "Item was updated" << std::endl;
   } else {
       std::cerr << outcome.GetError().GetMessage() << std::endl;
       return false;
```

```
    }

    return waitTableActive(tableName, dynamoClient);
}
```

Code that waits for the table to become active.

```
//! Query a newly created DynamoDB table until it is active.
/*!
  \sa waitTableActive()
  \param waitTableActive: The DynamoDB table's name.
  \param dynamoClient: A DynamoDB client.
  \return bool: Function succeeded.
*/
bool AwsDoc::DynamoDB::waitTableActive(const Aws::String &tableName,
                                       const Aws::DynamoDB::DynamoDBClient
 &dynamoClient) {

    // Repeatedly call DescribeTable until table is ACTIVE.
    const int MAX_QUERIES = 20;
    Aws::DynamoDB::Model::DescribeTableRequest request;
    request.SetTableName(tableName);

    int count = 0;
    while (count < MAX_QUERIES) {
        const Aws::DynamoDB::Model::DescribeTableOutcome &result =
 dynamoClient.DescribeTable(
                request);
        if (result.IsSuccess()) {
            Aws::DynamoDB::Model::TableStatus status =
 result.GetResult().GetTable().GetTableStatus();

            if (Aws::DynamoDB::Model::TableStatus::ACTIVE != status) {
                std::this_thread::sleep_for(std::chrono::seconds(1));
            }
            else {
                return true;
            }
        }
        else {
            std::cerr << "Error DynamoDB::waitTableActive "
                      << result.GetError().GetMessage() << std::endl;
```

```
            return false;
        }
        count++;
    }
    return false;
}
```

- For API details, see UpdateItem in *AWS SDK for C++ API Reference*.

CLI

AWS CLI

Example 1: To update an item in a table

The following `update-item` example updates an item in the `MusicCollection` table. It adds a new attribute (`Year`) and modifies the `AlbumTitle` attribute. All of the attributes in the item, as they appear after the update, are returned in the response.

```
aws dynamodb update-item \
    --table-name MusicCollection \
    --key file://key.json \
    --update-expression "SET #Y = :y, #AT = :t" \
    --expression-attribute-names file://expression-attribute-names.json \
    --expression-attribute-values file://expression-attribute-values.json  \
    --return-values ALL_NEW \
    --return-consumed-capacity TOTAL \
    --return-item-collection-metrics SIZE
```

Contents of `key.json`:

```
{
    "Artist": {"S": "Acme Band"},
    "SongTitle": {"S": "Happy Day"}
}
```

Contents of `expression-attribute-names.json`:

```
{
    "#Y":"Year", "#AT":"AlbumTitle"
```

```
}
```

Contents of `expression-attribute-values.json`:

```
{
    ":y":{"N": "2015"},
    ":t":{"S": "Louder Than Ever"}
}
```

Output:

```
{
    "Attributes": {
        "AlbumTitle": {
            "S": "Louder Than Ever"
        },
        "Awards": {
            "N": "10"
        },
        "Artist": {
            "S": "Acme Band"
        },
        "Year": {
            "N": "2015"
        },
        "SongTitle": {
            "S": "Happy Day"
        }
    },
    "ConsumedCapacity": {
        "TableName": "MusicCollection",
        "CapacityUnits": 3.0
    },
    "ItemCollectionMetrics": {
        "ItemCollectionKey": {
            "Artist": {
                "S": "Acme Band"
            }
        },
        "SizeEstimateRangeGB": [
            0.0,
            1.0
        ]
```

```
    }
}
```

For more information, see Writing an Item in the *Amazon DynamoDB Developer Guide*.

Example 2: To update an item conditionally

The following example updates an item in the `MusicCollection` table, but only if the existing item does not already have a `Year` attribute.

```
aws dynamodb update-item \
    --table-name MusicCollection \
    --key file://key.json \
    --update-expression "SET #Y = :y, #AT = :t" \
    --expression-attribute-names file://expression-attribute-names.json \
    --expression-attribute-values file://expression-attribute-values.json  \
    --condition-expression "attribute_not_exists(#Y)"
```

Contents of `key.json`:

```
{
    "Artist": {"S": "Acme Band"},
    "SongTitle": {"S": "Happy Day"}
}
```

Contents of `expression-attribute-names.json`:

```
{
    "#Y":"Year",
    "#AT":"AlbumTitle"
}
```

Contents of `expression-attribute-values.json`:

```
{
    ":y":{"N": "2015"},
    ":t":{"S": "Louder Than Ever"}
}
```

If the item already has a `Year` attribute, DynamoDB returns the following output.

```
An error occurred (ConditionalCheckFailedException) when calling the UpdateItem
 operation: The conditional request failed
```

For more information, see Writing an Item in the *Amazon DynamoDB Developer Guide*.

- For API details, see UpdateItem in *AWS CLI Command Reference*.

Go

SDK for Go V2

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import (
 "context"
 "errors"
 "log"
 "time"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/attributevalue"
 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/expression"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb/types"
)

// TableBasics encapsulates the Amazon DynamoDB service actions used in the
 examples.
// It contains a DynamoDB service client that is used to act on the specified
 table.
type TableBasics struct {
 DynamoDbClient *dynamodb.Client
 TableName      string
}
```

```
// UpdateMovie updates the rating and plot of a movie that already exists in the
// DynamoDB table. This function uses the `expression` package to build the
 update
// expression.
func (basics TableBasics) UpdateMovie(ctx context.Context, movie Movie)
 (map[string]map[string]interface{}, error) {
 var err error
 var response *dynamodb.UpdateItemOutput
 var attributeMap map[string]map[string]interface{}
 update := expression.Set(expression.Name("info.rating"),
 expression.Value(movie.Info["rating"]))
 update.Set(expression.Name("info.plot"), expression.Value(movie.Info["plot"]))
 expr, err := expression.NewBuilder().WithUpdate(update).Build()
 if err != nil {
  log.Printf("Couldn't build expression for update. Here's why: %v\n", err)
 } else {
  response, err = basics.DynamoDbClient.UpdateItem(ctx,
 &dynamodb.UpdateItemInput{
   TableName:                 aws.String(basics.TableName),
   Key:                       movie.GetKey(),
   ExpressionAttributeNames:  expr.Names(),
   ExpressionAttributeValues: expr.Values(),
   UpdateExpression:          expr.Update(),
   ReturnValues:              types.ReturnValueUpdatedNew,
  })
  if err != nil {
   log.Printf("Couldn't update movie %v. Here's why: %v\n", movie.Title, err)
  } else {
   err = attributevalue.UnmarshalMap(response.Attributes, &attributeMap)
   if err != nil {
    log.Printf("Couldn't unmarshall update response. Here's why: %v\n", err)
   }
  }
 }
 return attributeMap, err
}
```

Define a Movie struct that is used in this example.

```
import (
 "archive/zip"
 "bytes"
 "encoding/json"
 "fmt"
 "io"
 "log"
 "net/http"

 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/attributevalue"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb/types"
)

// Movie encapsulates data about a movie. Title and Year are the composite
 primary key
// of the movie in Amazon DynamoDB. Title is the sort key, Year is the partition
 key,
// and Info is additional data.
type Movie struct {
 Title string                 `dynamodbav:"title"`
 Year  int                    `dynamodbav:"year"`
 Info  map[string]interface{} `dynamodbav:"info"`
}

// GetKey returns the composite primary key of the movie in a format that can be
// sent to DynamoDB.
func (movie Movie) GetKey() map[string]types.AttributeValue {
 title, err := attributevalue.Marshal(movie.Title)
 if err != nil {
  panic(err)
 }
 year, err := attributevalue.Marshal(movie.Year)
 if err != nil {
  panic(err)
 }
 return map[string]types.AttributeValue{"title": title, "year": year}
}

// String returns the title, year, rating, and plot of a movie, formatted for the
 example.
func (movie Movie) String() string {
 return fmt.Sprintf("%v\n\tReleased: %v\n\tRating: %v\n\tPlot: %v\n",
  movie.Title, movie.Year, movie.Info["rating"], movie.Info["plot"])
}
```

- For API details, see UpdateItem in *AWS SDK for Go API Reference*.

Java

SDK for Java 2.x

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Updates an item in a table using DynamoDbClient.

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.AttributeAction;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.AttributeValueUpdate;
import software.amazon.awssdk.services.dynamodb.model.UpdateItemRequest;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import java.util.HashMap;

/**
 * Before running this Java V2 code example, set up your development
 * environment, including your credentials.
 *
 * For more information, see the following documentation topic:
 *
 * https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-
started.html
 *
 * To update an Amazon DynamoDB table using the AWS SDK for Java V2, its better
 * practice to use the
 * Enhanced Client, See the EnhancedModifyItem example.
 */
public class UpdateItem {
    public static void main(String[] args) {
        final String usage = """
```

```
                Usage:
                    <tableName> <key> <keyVal> <name> <updateVal>

                Where:
                    tableName - The Amazon DynamoDB table (for example, Music3).
                    key - The name of the key in the table (for example, Artist).
                    keyVal - The value of the key (for example, Famous Band).
                    name - The name of the column where the value is updated (for
example, Awards).
                    updateVal - The value used to update an item (for example,
14).
                 Example:
                    UpdateItem Music3 Artist Famous Band Awards 14
                """;

        if (args.length != 5) {
            System.out.println(usage);
            System.exit(1);
        }

        String tableName = args[0];
        String key = args[1];
        String keyVal = args[2];
        String name = args[3];
        String updateVal = args[4];

        Region region = Region.US_EAST_1;
        DynamoDbClient ddb = DynamoDbClient.builder()
                .region(region)
                .build();
        updateTableItem(ddb, tableName, key, keyVal, name, updateVal);
        ddb.close();
    }

    public static void updateTableItem(DynamoDbClient ddb,
            String tableName,
            String key,
            String keyVal,
            String name,
            String updateVal) {

        HashMap<String, AttributeValue> itemKey = new HashMap<>();
        itemKey.put(key, AttributeValue.builder()
```

```
                .s(keyVal)
                .build());

        HashMap<String, AttributeValueUpdate> updatedValues = new HashMap<>();
        updatedValues.put(name, AttributeValueUpdate.builder()
                .value(AttributeValue.builder().s(updateVal).build())
                .action(AttributeAction.PUT)
                .build());

        UpdateItemRequest request = UpdateItemRequest.builder()
                .tableName(tableName)
                .key(itemKey)
                .attributeUpdates(updatedValues)
                .build();

        try {
            ddb.updateItem(request);
        } catch (DynamoDbException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
        System.out.println("The Amazon DynamoDB table was updated!");
    }
}
```

- For API details, see UpdateItem in *AWS SDK for Java 2.x API Reference*.

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

This example uses the document client to simplify working with items in DynamoDB. For API details see UpdateCommand.

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
```

```
import { DynamoDBDocumentClient, UpdateCommand } from "@aws-sdk/lib-dynamodb";

const client = new DynamoDBClient({});
const docClient = DynamoDBDocumentClient.from(client);

export const main = async () => {
  const command = new UpdateCommand({
    TableName: "Dogs",
    Key: {
      Breed: "Labrador",
    },
    UpdateExpression: "set Color = :color",
    ExpressionAttributeValues: {
      ":color": "black",
    },
    ReturnValues: "ALL_NEW",
  });

  const response = await docClient.send(command);
  console.log(response);
  return response;
};
```

- For API details, see UpdateItem in *AWS SDK for JavaScript API Reference*.

Kotlin

SDK for Kotlin

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
suspend fun updateTableItem(
    tableNameVal: String,
    keyName: String,
    keyVal: String,
    name: String,
    updateVal: String,
```

```
) {
    val itemKey = mutableMapOf<String, AttributeValue>()
    itemKey[keyName] = AttributeValue.S(keyVal)

    val updatedValues = mutableMapOf<String, AttributeValueUpdate>()
    updatedValues[name] =
        AttributeValueUpdate {
            value = AttributeValue.S(updateVal)
            action = AttributeAction.Put
        }

    val request =
        UpdateItemRequest {
            tableName = tableNameVal
            key = itemKey
            attributeUpdates = updatedValues
        }

    DynamoDbClient.fromEnvironment { region = "us-east-1" }.use { ddb ->
        ddb.updateItem(request)
        println("Item in $tableNameVal was updated")
    }
}
```

- For API details, see UpdateItem in *AWS SDK for Kotlin API reference*.

PHP

SDK for PHP

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
        echo "What rating would you like to give {$movie['Item']['title']['S']}?
\n";
        $rating = 0;
        while (!is_numeric($rating) || intval($rating) != $rating || $rating < 1
 || $rating > 10) {
```

```
            $rating = testable_readline("Rating (1-10): ");
        }
        $service->updateItemAttributeByKey($tableName, $key, 'rating', 'N',
 $rating);

    public function updateItemAttributeByKey(
        string $tableName,
        array $key,
        string $attributeName,
        string $attributeType,
        string $newValue
    ) {
        $this->dynamoDbClient->updateItem([
            'Key' => $key['Item'],
            'TableName' => $tableName,
            'UpdateExpression' => "set #NV=:NV",
            'ExpressionAttributeNames' => [
                '#NV' => $attributeName,
            ],
            'ExpressionAttributeValues' => [
                ':NV' => [
                    $attributeType => $newValue
                ]
            ],
        ]);
    }
```

- For API details, see [UpdateItem](#) in *AWS SDK for PHP API Reference*.

PowerShell

Tools for PowerShell V4

Example 1: Sets the genre attribute to 'Rap' on the DynamoDB item with the partition key SongTitle and the sort key Artist.

```
$key = @{
    SongTitle = 'Somewhere Down The Road'
    Artist = 'No One You Know'
} | ConvertTo-DDBItem

$updateDdbItem = @{
```

```
    TableName = 'Music'
    Key = $key
    UpdateExpression = 'set Genre = :val1'
    ExpressionAttributeValue = (@{
        ':val1' = ([Amazon.DynamoDBv2.Model.AttributeValue]'Rap')
    })
}
Update-DDBItem @updateDdbItem
```

Output:

```
Name                           Value
----                           -----
Genre                          Rap
```

- For API details, see UpdateItem in *AWS Tools for PowerShell Cmdlet Reference (V4)*.

Tools for PowerShell V5

Example 1: Sets the genre attribute to 'Rap' on the DynamoDB item with the partition key SongTitle and the sort key Artist.

```
$key = @{
    SongTitle = 'Somewhere Down The Road'
    Artist = 'No One You Know'
} | ConvertTo-DDBItem

$updateDdbItem = @{
    TableName = 'Music'
    Key = $key
    UpdateExpression = 'set Genre = :val1'
    ExpressionAttributeValue = (@{
        ':val1' = ([Amazon.DynamoDBv2.Model.AttributeValue]'Rap')
    })
}
Update-DDBItem @updateDdbItem
```

Output:

```
Name                           Value
----                           -----
Genre                          Rap
```

- For API details, see UpdateItem in *AWS Tools for PowerShell Cmdlet Reference (V5)*.

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Update an item by using an update expression.

```
class Movies:
    """Encapsulates an Amazon DynamoDB table of movie data.

    Example data structure for a movie record in this table:
        {
            "year": 1999,
            "title": "For Love of the Game",
            "info": {
                "directors": ["Sam Raimi"],
                "release_date": "1999-09-15T00:00:00Z",
                "rating": 6.3,
                "plot": "A washed up pitcher flashes through his career.",
                "rank": 4987,
                "running_time_secs": 8220,
                "actors": [
                    "Kevin Costner",
                    "Kelly Preston",
                    "John C. Reilly"
                ]
            }
        }
    """

    def __init__(self, dyn_resource):
        """
        :param dyn_resource: A Boto3 DynamoDB resource.
        """
        self.dyn_resource = dyn_resource
```

```
        # The table variable is set during the scenario in the call to
        # 'exists' if the table exists. Otherwise, it is set by 'create_table'.
        self.table = None


    def update_movie(self, title, year, rating, plot):
        """
        Updates rating and plot data for a movie in the table.

        :param title: The title of the movie to update.
        :param year: The release year of the movie to update.
        :param rating: The updated rating to the give the movie.
        :param plot: The updated plot summary to give the movie.
        :return: The fields that were updated, with their new values.
        """
        try:
            response = self.table.update_item(
                Key={"year": year, "title": title},
                UpdateExpression="set info.rating=:r, info.plot=:p",
                ExpressionAttributeValues={":r": Decimal(str(rating)), ":p":
 plot},
                ReturnValues="UPDATED_NEW",
            )
        except ClientError as err:
            logger.error(
                "Couldn't update movie %s in table %s. Here's why: %s: %s",
                title,
                self.table.name,
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
        else:
            return response["Attributes"]
```

Update an item by using an update expression that includes an arithmetic operation.

```
class UpdateQueryWrapper:
    def __init__(self, table):
        self.table = table
```

```
    def update_rating(self, title, year, rating_change):
        """
        Updates the quality rating of a movie in the table by using an arithmetic
        operation in the update expression. By specifying an arithmetic
operation,
        you can adjust a value in a single request, rather than first getting its
        value and then setting its new value.

        :param title: The title of the movie to update.
        :param year: The release year of the movie to update.
        :param rating_change: The amount to add to the current rating for the
movie.
        :return: The updated rating.
        """
        try:
            response = self.table.update_item(
                Key={"year": year, "title": title},
                UpdateExpression="set info.rating = info.rating + :val",
                ExpressionAttributeValues={":val": Decimal(str(rating_change))},
                ReturnValues="UPDATED_NEW",
            )
        except ClientError as err:
            logger.error(
                "Couldn't update movie %s in table %s. Here's why: %s: %s",
                title,
                self.table.name,
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
        else:
            return response["Attributes"]
```

Update an item only when it meets certain conditions.

```
class UpdateQueryWrapper:
    def __init__(self, table):
        self.table = table
```

```
    def remove_actors(self, title, year, actor_threshold):
        """
        Removes an actor from a movie, but only when the number of actors is
greater
        than a specified threshold. If the movie does not list more than the
threshold,
        no actors are removed.

        :param title: The title of the movie to update.
        :param year: The release year of the movie to update.
        :param actor_threshold: The threshold of actors to check.
        :return: The movie data after the update.
        """
        try:
            response = self.table.update_item(
                Key={"year": year, "title": title},
                UpdateExpression="remove info.actors[0]",
                ConditionExpression="size(info.actors) > :num",
                ExpressionAttributeValues={":num": actor_threshold},
                ReturnValues="ALL_NEW",
            )
        except ClientError as err:
            if err.response["Error"]["Code"] ==
"ConditionalCheckFailedException":
                logger.warning(
                    "Didn't update %s because it has fewer than %s actors.",
                    title,
                    actor_threshold + 1,
                )
            else:
                logger.error(
                    "Couldn't update movie %s. Here's why: %s: %s",
                    title,
                    err.response["Error"]["Code"],
                    err.response["Error"]["Message"],
                )
            raise
        else:
            return response["Attributes"]
```

- For API details, see UpdateItem in *AWS SDK for Python (Boto3) API Reference*.

Ruby

SDK for Ruby

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
class DynamoDBBasics
  attr_reader :dynamo_resource, :table

  def initialize(table_name)
    client = Aws::DynamoDB::Client.new(region: 'us-east-1')
    @dynamo_resource = Aws::DynamoDB::Resource.new(client: client)
    @table = @dynamo_resource.table(table_name)
  end

  # Updates rating and plot data for a movie in the table.
  #
  # @param movie [Hash] The title, year, plot, rating of the movie.
  def update_item(movie)
    response = @table.update_item(
      key: { 'year' => movie[:year], 'title' => movie[:title] },
      update_expression: 'set info.rating=:r',
      expression_attribute_values: { ':r' => movie[:rating] },
      return_values: 'UPDATED_NEW'
    )
  rescue Aws::DynamoDB::Errors::ServiceError => e
    puts("Couldn't update movie #{movie[:title]} (#{movie[:year]}) in table
 #{@table.name}\n")
    puts("\t#{e.code}: #{e.message}")
    raise
  else
    response.attributes
  end
```

- For API details, see UpdateItem in *AWS SDK for Ruby API Reference*.

SAP ABAP

SDK for SAP ABAP

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    TRY.
        oo_output = lo_dyn->updateitem(
          iv_tablename        = iv_table_name
          it_key              = it_item_key
          it_attributeupdates = it_attribute_updates ).
        MESSAGE '1 item updated in DynamoDB Table' && iv_table_name TYPE 'I'.
      CATCH /aws1/cx_dyncondalcheckfaile00.
        MESSAGE 'A condition specified in the operation could not be evaluated.'
TYPE 'E'.
      CATCH /aws1/cx_dynresourcenotfoundex.
        MESSAGE 'The table or index does not exist' TYPE 'E'.
      CATCH /aws1/cx_dyntransactconflictex.
        MESSAGE 'Another transaction is using the item' TYPE 'E'.
    ENDTRY.
```

- For API details, see UpdateItem in *AWS SDK for SAP ABAP API reference*.

Swift

SDK for Swift

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import AWSDynamoDB
```

```
    /// Update the specified movie with new `rating` and `plot` information.
    ///
    /// - Parameters:
    ///   - title: The title of the movie to update.
    ///   - year: The release year of the movie to update.
    ///   - rating: The new rating for the movie.
    ///   - plot: The new plot summary string for the movie.
    ///
    /// - Returns: An array of mappings of attribute names to their new
    ///   listing each item actually changed. Items that didn't need to change
    ///   aren't included in this list. `nil` if no changes were made.
    ///
    func update(title: String, year: Int, rating: Double? = nil, plot: String? =
nil) async throws
        -> [Swift.String: DynamoDBClientTypes.AttributeValue]?
    {
        do {
            guard let client = self.ddbClient else {
                throw MoviesError.UninitializedClient
            }

            // Build the update expression and the list of expression attribute
            // values. Include only the information that's changed.

            var expressionParts: [String] = []
            var attrValues: [Swift.String: DynamoDBClientTypes.AttributeValue] =
[:]

            if rating != nil {
                expressionParts.append("info.rating=:r")
                attrValues[":r"] = .n(String(rating!))
            }
            if plot != nil {
                expressionParts.append("info.plot=:p")
                attrValues[":p"] = .s(plot!)
            }
            let expression = "set \(expressionParts.joined(separator: ", "))"

            let input = UpdateItemInput(
                // Create substitution tokens for the attribute values, to ensure
                // no conflicts in expression syntax.
                expressionAttributeValues: attrValues,
```

```
                // The key identifying the movie to update consists of the
release
                // year and title.
                key: [
                    "year": .n(String(year)),
                    "title": .s(title)
                ],
                returnValues: .updatedNew,
                tableName: self.tableName,
                updateExpression: expression
            )
            let output = try await client.updateItem(input: input)

            guard let attributes: [Swift.String:
DynamoDBClientTypes.AttributeValue] = output.attributes else {
                throw MoviesError.InvalidAttributes
            }
            return attributes
        } catch {
            print("ERROR: update:", dump(error))
            throw error
        }
    }
```

- For API details, see UpdateItem in *AWS SDK for Swift API reference*.

For more DynamoDB examples, see Code examples for DynamoDB using AWS SDKs.

To query the data in the `Music` table, proceed to Step 5: Query data in a DynamoDB table.

Step 5: Query data in a DynamoDB table

In this step, you query the data that you wrote to the `Music` table in the section called "Step 2: Write data" by specifying `Artist`. This will display all songs that are associated with the partition key: `Artist`.

For more information about query operations, see Querying tables in DynamoDB.

AWS Management Console

Follow these steps to use the DynamoDB console to query data in the `Music` table.

1. Open the DynamoDB console at https://console.aws.amazon.com/dynamodb/.
2. In the left navigation pane, choose **Tables**.
3. Choose the **Music** table from the table list.
4. Choose **Explore table items**.
5. In **Scan or query items**, make sure that **Query** is selected.
6. For **Partition key**, enter **`Acme Band`**, and then choose **Run**.

AWS CLI

The following AWS CLI example queries an item in the `Music` table. You can do this either through the DynamoDB API or PartiQL, a SQL-compatible query language for DynamoDB.

DynamoDB API

You query an item through the DynamoDB API by using `query` and providing the partition key.

Linux

```
aws dynamodb query \
    --table-name Music \
    --key-condition-expression "Artist = :name" \
    --expression-attribute-values  '{":name":{"S":"Acme Band"}}'
```

Windows CMD

```
aws dynamodb query ^
    --table-name Music ^
    --key-condition-expression "Artist = :name" ^
    --expression-attribute-values  "{\":name\":{\"S\":\"Acme Band\"}}"
```

Using `query` returns all the songs associated with this particular `Artist`.

```
{
    "Items": [
```

```
        {
            "AlbumTitle": {
                "S": "Updated Album Title"
            },
            "Awards": {
                "N": "10"
            },
            "Artist": {
                "S": "Acme Band"
            },
            "SongTitle": {
                "S": "Happy Day"
            }
        },
        {
            "AlbumTitle": {
                "S": "Another Album Title"
            },
            "Awards": {
                "N": "8"
            },
            "Artist": {
                "S": "Acme Band"
            },
            "SongTitle": {
                "S": "PartiQL Rocks"
            }
        }
    ],
    "Count": 2,
    "ScannedCount": 2,
    "ConsumedCapacity": null
}
```

PartiQL for DynamoDB

You query an item through PartiQL by using the `Select` statement and providing the partition key.

Linux

```
aws dynamodb execute-statement --statement "SELECT * FROM Music   \
                                            WHERE Artist='Acme Band'"
```

Windows CMD

```
aws dynamodb execute-statement --statement "SELECT * FROM Music WHERE Artist='Acme
 Band'"
```

Using the `Select` statement in this way returns all the songs associated with this particular `Artist`.

```
{
    "Items": [
        {
            "AlbumTitle": {
                "S": "Updated Album Title"
            },
            "Awards": {
                "S": "10"
            },
            "Artist": {
                "S": "Acme Band"
            },
            "SongTitle": {
                "S": "Happy Day"
            }
        },
        {
            "AlbumTitle": {
                "S": "Another Album Title"
            },
            "Awards": {
                "S": "8"
            },
            "Artist": {
                "S": "Acme Band"
            },
            "SongTitle": {
                "S": "PartiQL Rocks"
            }
        }
    ]
}
```

For more information about querying data with PartiQL, see PartiQL select statements.

AWS SDK

The following code examples show how to query a DynamoDB table using an AWS SDK.

.NET

SDK for .NET (v4)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
/// <summary>
/// Queries the table for movies released in a particular year and
/// then displays the information for the movies returned.
/// </summary>
/// <param name="tableName">The name of the table to query.</param>
/// <param name="year">The release year for which we want to
/// view movies.</param>
/// <returns>The number of movies that match the query.</returns>
public async Task<int> QueryMoviesAsync(string tableName, int year)
{
    try
    {
        var movieTable = new TableBuilder(_amazonDynamoDB, tableName)
            .AddHashKey("year", DynamoDBEntryType.Numeric)
            .AddRangeKey("title", DynamoDBEntryType.String)
            .Build();

        var filter = new QueryFilter("year", QueryOperator.Equal, year);

        Console.WriteLine("\nFind movies released in: {year}:");

        var config = new QueryOperationConfig()
        {
            Limit = 10, // 10 items per page.
            Select = SelectValues.SpecificAttributes,
            AttributesToGet = new List<string>
            {
```

```
                    "title",
                    "year",
                },
                ConsistentRead = true,
                Filter = filter,
            };

            // Value used to track how many movies match the
            // supplied criteria.
            var moviesFound = 0;

            var search = movieTable.Query(config);
            do
            {
                var movieList = await search.GetNextSetAsync();
                moviesFound += movieList.Count;

                foreach (var movie in movieList)
                {
                    DisplayDocument(movie);
                }
            }
            while (!search.IsDone);

            return moviesFound;
        }
        catch (ResourceNotFoundException ex)
        {
            Console.WriteLine($"Table {tableName} was not found. {ex.Message}");
            return 0;
        }
        catch (AmazonDynamoDBException ex)
        {
            Console.WriteLine($"An Amazon DynamoDB error occurred while querying
movies. {ex.Message}");
            throw;
        }
        catch (Exception ex)
        {
            Console.WriteLine($"An error occurred while querying movies.
{ex.Message}");
            throw;
        }
    }
```

- For API details, see Query in *AWS SDK for .NET API Reference*.

Bash

AWS CLI with Bash script

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
##############################################################################
# function dynamodb_query
#
# This function queries a DynamoDB table.
#
# Parameters:
#       -n table_name  -- The name of the table.
#       -k key_condition_expression -- The key condition expression.
#       -a attribute_names -- Path to JSON file containing the attribute names.
#       -v attribute_values -- Path to JSON file containing the attribute values.
#       [-p projection_expression]  -- Optional projection expression.
#
#  Returns:
#       The items as json output.
#  And:
#       0 - If successful.
#       1 - If it fails.
##############################################################################
function dynamodb_query() {
  local table_name key_condition_expression attribute_names attribute_values
 projection_expression response
  local option OPTARG # Required to use getopts command in a function.

  # ######################################
  # Function usage explanation
  ######################################
  function usage() {
```

```
   echo "function dynamodb_query"
   echo "Query a DynamoDB table."
   echo " -n table_name  -- The name of the table."
   echo " -k key_condition_expression -- The key condition expression."
   echo " -a attribute_names -- Path to JSON file containing the attribute
names."
   echo " -v attribute_values -- Path to JSON file containing the attribute
values."
   echo " [-p projection_expression]  -- Optional projection expression."
   echo ""
 }

 while getopts "n:k:a:v:p:h" option; do
   case "${option}" in
     n) table_name="${OPTARG}" ;;
     k) key_condition_expression="${OPTARG}" ;;
     a) attribute_names="${OPTARG}" ;;
     v) attribute_values="${OPTARG}" ;;
     p) projection_expression="${OPTARG}" ;;
     h)
       usage
       return 0
       ;;
     \?)
       echo "Invalid parameter"
       usage
       return 1
       ;;
   esac
 done
 export OPTIND=1

 if [[ -z "$table_name" ]]; then
   errecho "ERROR: You must provide a table name with the -n parameter."
   usage
   return 1
 fi

 if [[ -z "$key_condition_expression" ]]; then
   errecho "ERROR: You must provide a key condition expression with the -k
parameter."
   usage
   return 1
 fi
```

```
  if [[ -z "$attribute_names" ]]; then
    errecho "ERROR: You must provide a attribute names with the -a parameter."
    usage
    return 1
  fi

  if [[ -z "$attribute_values" ]]; then
    errecho "ERROR: You must provide a attribute values with the -v parameter."
    usage
    return 1
  fi

  if [[ -z "$projection_expression" ]]; then
    response=$(aws dynamodb query \
      --table-name "$table_name" \
      --key-condition-expression "$key_condition_expression" \
      --expression-attribute-names file://"$attribute_names" \
      --expression-attribute-values file://"$attribute_values")
  else
    response=$(aws dynamodb query \
      --table-name "$table_name" \
      --key-condition-expression "$key_condition_expression" \
      --expression-attribute-names file://"$attribute_names" \
      --expression-attribute-values file://"$attribute_values" \
      --projection-expression "$projection_expression")
  fi

  local error_code=${?}

  if [[ $error_code -ne 0 ]]; then
    aws_cli_error_log $error_code
    errecho "ERROR: AWS reports query operation failed.$response"
    return 1
  fi

  echo "$response"

  return 0
}
```

The utility functions used in this example.

```
#####################################################################################
# function errecho
#
# This function outputs everything sent to it to STDERR (standard error output).
#####################################################################################
function errecho() {
  printf "%s\n" "$*" 1>&2
}

#####################################################################################
# function aws_cli_error_log()
#
# This function is used to log the error messages from the AWS CLI.
#
# See https://docs.aws.amazon.com/cli/latest/topic/return-codes.html#cli-aws-
help-return-codes.
#
# The function expects the following argument:
#         $1 - The error code returned by the AWS CLI.
#
#  Returns:
#          0: - Success.
#
#####################################################################################
function aws_cli_error_log() {
  local err_code=$1
  errecho "Error code : $err_code"
  if [ "$err_code" == 1 ]; then
    errecho "  One or more S3 transfers failed."
  elif [ "$err_code" == 2 ]; then
    errecho "  Command line failed to parse."
  elif [ "$err_code" == 130 ]; then
    errecho "  Process received SIGINT."
  elif [ "$err_code" == 252 ]; then
    errecho "  Command syntax invalid."
  elif [ "$err_code" == 253 ]; then
    errecho "  The system environment or configuration was invalid."
  elif [ "$err_code" == 254 ]; then
    errecho "  The service returned an error."
  elif [ "$err_code" == 255 ]; then
    errecho "  255 is a catch-all error."
  fi
```

```
  return 0
}
```

- For API details, see Query in *AWS CLI Command Reference*.

C++

SDK for C++

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
//! Perform a query on an Amazon DynamoDB Table and retrieve items.
/*!
  \sa queryItem()
  \param tableName: The table name.
  \param partitionKey: The partition key.
  \param partitionValue: The value for the partition key.
  \param projectionExpression: The projections expression, which is ignored if
 empty.
  \param clientConfiguration: AWS client configuration.
  \return bool: Function succeeded.
  */

/*
 * The partition key attribute is searched with the specified value. By default,
 all fields and values
 * contained in the item are returned. If an optional projection expression is
 * specified on the command line, only the specified fields and values are
 * returned.
 */

bool AwsDoc::DynamoDB::queryItems(const Aws::String &tableName,
                                  const Aws::String &partitionKey,
                                  const Aws::String &partitionValue,
                                  const Aws::String &projectionExpression,
                                  const Aws::Client::ClientConfiguration
 &clientConfiguration) {
```

```
    Aws::DynamoDB::DynamoDBClient dynamoClient(clientConfiguration);
    Aws::DynamoDB::Model::QueryRequest request;

    request.SetTableName(tableName);

    if (!projectionExpression.empty()) {
        request.SetProjectionExpression(projectionExpression);
    }

    // Set query key condition expression.
    request.SetKeyConditionExpression(partitionKey + "= :valueToMatch");

    // Set Expression AttributeValues.
    Aws::Map<Aws::String, Aws::DynamoDB::Model::AttributeValue> attributeValues;
    attributeValues.emplace(":valueToMatch", partitionValue);

    request.SetExpressionAttributeValues(attributeValues);

    bool result = true;

    // "exclusiveStartKey" is used for pagination.
    Aws::Map<Aws::String, Aws::DynamoDB::Model::AttributeValue>
exclusiveStartKey;
    do {
        if (!exclusiveStartKey.empty()) {
            request.SetExclusiveStartKey(exclusiveStartKey);
            exclusiveStartKey.clear();
        }
        // Perform Query operation.
        const Aws::DynamoDB::Model::QueryOutcome &outcome =
dynamoClient.Query(request);
        if (outcome.IsSuccess()) {
            // Reference the retrieved items.
            const Aws::Vector<Aws::Map<Aws::String,
Aws::DynamoDB::Model::AttributeValue>> &items = outcome.GetResult().GetItems();
            if (!items.empty()) {
                std::cout << "Number of items retrieved from Query: " <<
items.size()
                          << std::endl;
                // Iterate each item and print.
                for (const auto &item: items) {
                    std::cout
                            <<
"*************************************************"
```

```
                            << std::endl;
                    // Output each retrieved field and its value.
                    for (const auto &i: item)
                        std::cout << i.first << ": " << i.second.GetS() <<
 std::endl;
                }
            }
            else {
                std::cout << "No item found in table: " << tableName <<
 std::endl;
            }

            exclusiveStartKey = outcome.GetResult().GetLastEvaluatedKey();
        }
        else {
            std::cerr << "Failed to Query items: " <<
 outcome.GetError().GetMessage();
            result = false;
            break;
        }
    } while (!exclusiveStartKey.empty());

    return result;
}
```

- For API details, see Query in *AWS SDK for C++ API Reference*.

CLI

AWS CLI

Example 1: To query a table

The following `query` example queries items in the `MusicCollection` table. The table has a hash-and-range primary key (`Artist` and `SongTitle`), but this query only specifies the hash key value. It returns song titles by the artist named "No One You Know".

```
aws dynamodb query \
    --table-name MusicCollection \
    --projection-expression "SongTitle" \
    --key-condition-expression "Artist = :v1" \
    --expression-attribute-values file://expression-attributes.json \
```

```
    --return-consumed-capacity TOTAL
```

Contents of `expression-attributes.json`:

```
{
    ":v1": {"S": "No One You Know"}
}
```

Output:

```
{
    "Items": [
        {
            "SongTitle": {
                "S": "Call Me Today"
            },
            "SongTitle": {
                "S": "Scared of My Shadow"
            }
        }
    ],
    "Count": 2,
    "ScannedCount": 2,
    "ConsumedCapacity": {
        "TableName": "MusicCollection",
        "CapacityUnits": 0.5
    }
}
```

For more information, see Working with Queries in DynamoDB in the *Amazon DynamoDB Developer Guide*.

Example 2: To query a table using strongly consistent reads and traverse the index in descending order

The following example performs the same query as the first example, but returns results in reverse order and uses strongly consistent reads.

```
aws dynamodb query \
    --table-name MusicCollection \
    --projection-expression "SongTitle" \
    --key-condition-expression "Artist = :v1" \
```

```
    --expression-attribute-values file://expression-attributes.json \
    --consistent-read \
    --no-scan-index-forward \
    --return-consumed-capacity TOTAL
```

Contents of `expression-attributes.json`:

```
{
    ":v1": {"S": "No One You Know"}
}
```

Output:

```
{
    "Items": [
        {
            "SongTitle": {
                "S": "Scared of My Shadow"
            }
        },
        {
            "SongTitle": {
                "S": "Call Me Today"
            }
        }
    ],
    "Count": 2,
    "ScannedCount": 2,
    "ConsumedCapacity": {
        "TableName": "MusicCollection",
        "CapacityUnits": 1.0
    }
}
```

For more information, see Working with Queries in DynamoDB in the *Amazon DynamoDB Developer Guide*.

Example 3: To filter out specific results

The following example queries the `MusicCollection` but excludes results with specific values in the `AlbumTitle` attribute. Note that this does not affect the `ScannedCount` or `ConsumedCapacity`, because the filter is applied after the items have been read.

```
aws dynamodb query \
    --table-name MusicCollection \
    --key-condition-expression "#n1 = :v1" \
    --filter-expression "NOT (#n2 IN (:v2, :v3))" \
    --expression-attribute-names file://names.json \
    --expression-attribute-values file://values.json \
    --return-consumed-capacity TOTAL
```

Contents of `values.json`:

```
{
    ":v1": {"S": "No One You Know"},
    ":v2": {"S": "Blue Sky Blues"},
    ":v3": {"S": "Greatest Hits"}
}
```

Contents of `names.json`:

```
{
    "#n1": "Artist",
    "#n2": "AlbumTitle"
}
```

Output:

```
{
    "Items": [
        {
            "AlbumTitle": {
                "S": "Somewhat Famous"
            },
            "Artist": {
                "S": "No One You Know"
            },
            "SongTitle": {
                "S": "Call Me Today"
            }
        }
    ],
    "Count": 1,
    "ScannedCount": 2,
```

```
    "ConsumedCapacity": {
        "TableName": "MusicCollection",
        "CapacityUnits": 0.5
    }
}
```

For more information, see Working with Queries in DynamoDB in the *Amazon DynamoDB Developer Guide*.

Example 4: To retrieve only an item count

The following example retrieves a count of items matching the query, but does not retrieve any of the items themselves.

```
aws dynamodb query \
    --table-name MusicCollection \
    --select COUNT \
    --key-condition-expression "Artist = :v1" \
    --expression-attribute-values file://expression-attributes.json
```

Contents of `expression-attributes.json`:

```
{
    ":v1": {"S": "No One You Know"}
}
```

Output:

```
{
    "Count": 2,
    "ScannedCount": 2,
    "ConsumedCapacity": null
}
```

For more information, see Working with Queries in DynamoDB in the *Amazon DynamoDB Developer Guide*.

Example 5: To query an index

The following example queries the local secondary index `AlbumTitleIndex`. The query returns all attributes from the base table that have been projected into the local secondary

index. Note that when querying a local secondary index or global secondary index, you must also provide the name of the base table using the `table-name` parameter.

```
aws dynamodb query \
    --table-name MusicCollection \
    --index-name AlbumTitleIndex \
    --key-condition-expression "Artist = :v1" \
    --expression-attribute-values file://expression-attributes.json \
    --select ALL_PROJECTED_ATTRIBUTES \
    --return-consumed-capacity INDEXES
```

Contents of `expression-attributes.json`:

```
{
    ":v1": {"S": "No One You Know"}
}
```

Output:

```
{
    "Items": [
        {
            "AlbumTitle": {
                "S": "Blue Sky Blues"
            },
            "Artist": {
                "S": "No One You Know"
            },
            "SongTitle": {
                "S": "Scared of My Shadow"
            }
        },
        {
            "AlbumTitle": {
                "S": "Somewhat Famous"
            },
            "Artist": {
                "S": "No One You Know"
            },
            "SongTitle": {
                "S": "Call Me Today"
            }
```

```
        }
    ],
    "Count": 2,
    "ScannedCount": 2,
    "ConsumedCapacity": {
        "TableName": "MusicCollection",
        "CapacityUnits": 0.5,
        "Table": {
            "CapacityUnits": 0.0
        },
        "LocalSecondaryIndexes": {
            "AlbumTitleIndex": {
                "CapacityUnits": 0.5
            }
        }
    }
}
```

For more information, see Working with Queries in DynamoDB in the *Amazon DynamoDB Developer Guide*.

- For API details, see Query in *AWS CLI Command Reference*.

Go

SDK for Go V2

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import (
 "context"
 "errors"
 "log"
 "time"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/attributevalue"
```

```
 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/expression"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb/types"
)

// TableBasics encapsulates the Amazon DynamoDB service actions used in the
 examples.
// It contains a DynamoDB service client that is used to act on the specified
 table.
type TableBasics struct {
 DynamoDbClient *dynamodb.Client
 TableName      string
}



// Query gets all movies in the DynamoDB table that were released in the
 specified year.
// The function uses the `expression` package to build the key condition
 expression
// that is used in the query.
func (basics TableBasics) Query(ctx context.Context, releaseYear int) ([]Movie,
 error) {
 var err error
 var response *dynamodb.QueryOutput
 var movies []Movie
 keyEx := expression.Key("year").Equal(expression.Value(releaseYear))
 expr, err := expression.NewBuilder().WithKeyCondition(keyEx).Build()
 if err != nil {
  log.Printf("Couldn't build expression for query. Here's why: %v\n", err)
 } else {
  queryPaginator := dynamodb.NewQueryPaginator(basics.DynamoDbClient,
 &dynamodb.QueryInput{
   TableName:                 aws.String(basics.TableName),
   ExpressionAttributeNames:  expr.Names(),
   ExpressionAttributeValues: expr.Values(),
   KeyConditionExpression:    expr.KeyCondition(),
  })
  for queryPaginator.HasMorePages() {
   response, err = queryPaginator.NextPage(ctx)
   if err != nil {
    log.Printf("Couldn't query for movies released in %v. Here's why: %v\n",
 releaseYear, err)
    break
```

```
   } else {
    var moviePage []Movie
    err = attributevalue.UnmarshalListOfMaps(response.Items, &moviePage)
    if err != nil {
     log.Printf("Couldn't unmarshal query response. Here's why: %v\n", err)
     break
    } else {
     movies = append(movies, moviePage...)
    }
   }
  }
 }
 return movies, err
}
```

Define a Movie struct that is used in this example.

```
import (
 "archive/zip"
 "bytes"
 "encoding/json"
 "fmt"
 "io"
 "log"
 "net/http"

 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/attributevalue"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb/types"
)

// Movie encapsulates data about a movie. Title and Year are the composite
 primary key
// of the movie in Amazon DynamoDB. Title is the sort key, Year is the partition
 key,
// and Info is additional data.
type Movie struct {
 Title string                `dynamodbav:"title"`
 Year  int                   `dynamodbav:"year"`
 Info  map[string]interface{} `dynamodbav:"info"`
}
```

```
// GetKey returns the composite primary key of the movie in a format that can be
// sent to DynamoDB.
func (movie Movie) GetKey() map[string]types.AttributeValue {
 title, err := attributevalue.Marshal(movie.Title)
 if err != nil {
  panic(err)
 }
 year, err := attributevalue.Marshal(movie.Year)
 if err != nil {
  panic(err)
 }
 return map[string]types.AttributeValue{"title": title, "year": year}
}

// String returns the title, year, rating, and plot of a movie, formatted for the
 example.
func (movie Movie) String() string {
 return fmt.Sprintf("%v\n\tReleased: %v\n\tRating: %v\n\tPlot: %v\n",
  movie.Title, movie.Year, movie.Info["rating"], movie.Info["plot"])
}
```

- For API details, see Query in *AWS SDK for Go API Reference*.

Java

SDK for Java 2.x

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Queries a table by using DynamoDbClient.

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
```

```
import software.amazon.awssdk.services.dynamodb.model.QueryRequest;
import software.amazon.awssdk.services.dynamodb.model.QueryResponse;
import java.util.HashMap;

/**
 * Before running this Java V2 code example, set up your development
 * environment, including your credentials.
 *
 * For more information, see the following documentation topic:
 *
 * https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-
started.html
 *
 * To query items from an Amazon DynamoDB table using the AWS SDK for Java V2,
 * its better practice to use the
 * Enhanced Client. See the EnhancedQueryRecords example.
 */
public class Query {
    public static void main(String[] args) {
        final String usage = """

                Usage:
                    <tableName> <partitionKeyName> <partitionKeyVal>

                Where:
                    tableName - The Amazon DynamoDB table to put the item in (for
 example, Music3).
                    partitionKeyName - The partition key name of the Amazon
 DynamoDB table (for example, Artist).
                    partitionKeyVal - The value of the partition key that should
 match (for example, Famous Band).
                """;

        if (args.length != 3) {
            System.out.println(usage);
            System.exit(1);
        }

        String tableName = args[0];
        String partitionKeyName = args[1];
        String partitionKeyVal = args[2];

        // For more information about an alias, see:
```

```
        // https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/
Expressions.ExpressionAttributeNames.html
        String partitionAlias = "#a";

        System.out.format("Querying %s", tableName);
        System.out.println("");
        Region region = Region.US_EAST_1;
        DynamoDbClient ddb = DynamoDbClient.builder()
                .region(region)
                .build();

        int count = queryTable(ddb, tableName, partitionKeyName, partitionKeyVal,
 partitionAlias);
        System.out.println("There were " + count + "  record(s) returned");
        ddb.close();
    }

    public static int queryTable(DynamoDbClient ddb, String tableName, String
 partitionKeyName, String partitionKeyVal,
            String partitionAlias) {
        // Set up an alias for the partition key name in case it's a reserved
 word.
        HashMap<String, String> attrNameAlias = new HashMap<String, String>();
        attrNameAlias.put(partitionAlias, partitionKeyName);

        // Set up mapping of the partition name with the value.
        HashMap<String, AttributeValue> attrValues = new HashMap<>();
        attrValues.put(":" + partitionKeyName, AttributeValue.builder()
                .s(partitionKeyVal)
                .build());

        QueryRequest queryReq = QueryRequest.builder()
                .tableName(tableName)
                .keyConditionExpression(partitionAlias + " = :" +
 partitionKeyName)
                .expressionAttributeNames(attrNameAlias)
                .expressionAttributeValues(attrValues)
                .build();

        try {
            QueryResponse response = ddb.query(queryReq);
            return response.count();

        } catch (DynamoDbException e) {
```

```
            System.err.println(e.getMessage());
            System.exit(1);
        }
        return -1;
    }
}
```

Queries a table by using `DynamoDbClient` and a secondary index.

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.QueryRequest;
import software.amazon.awssdk.services.dynamodb.model.QueryResponse;
import java.util.HashMap;
import java.util.Map;

/**
 * Before running this Java V2 code example, set up your development
 * environment, including your credentials.
 *
 * For more information, see the following documentation topic:
 *
 * https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-
started.html
 *
 * Create the Movies table by running the Scenario example and loading the Movie
 * data from the JSON file. Next create a secondary
 * index for the Movies table that uses only the year column. Name the index
 * **year-index**. For more information, see:
 *
 * https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/GSI.html
 */
public class QueryItemsUsingIndex {
    public static void main(String[] args) {
        String tableName = "Movies";
        Region region = Region.US_EAST_1;
        DynamoDbClient ddb = DynamoDbClient.builder()
                .region(region)
                .build();
```

```
        queryIndex(ddb, tableName);
        ddb.close();
    }

    public static void queryIndex(DynamoDbClient ddb, String tableName) {
        try {
            Map<String, String> expressionAttributesNames = new HashMap<>();
            expressionAttributesNames.put("#year", "year");
            Map<String, AttributeValue> expressionAttributeValues = new
 HashMap<>();
            expressionAttributeValues.put(":yearValue",
 AttributeValue.builder().n("2013").build());

            QueryRequest request = QueryRequest.builder()
                    .tableName(tableName)
                    .indexName("year-index")
                    .keyConditionExpression("#year = :yearValue")
                    .expressionAttributeNames(expressionAttributesNames)
                    .expressionAttributeValues(expressionAttributeValues)
                    .build();

            System.out.println("=== Movie Titles ===");
            QueryResponse response = ddb.query(request);
            response.items()
                    .forEach(movie ->
 System.out.println(movie.get("title").s()));

        } catch (DynamoDbException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }
}
```

- For API details, see Query in *AWS SDK for Java 2.x API Reference*.

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

This example uses the document client to simplify working with items in DynamoDB. For API details see QueryCommand.

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
import { QueryCommand, DynamoDBDocumentClient } from "@aws-sdk/lib-dynamodb";

const client = new DynamoDBClient({});
const docClient = DynamoDBDocumentClient.from(client);

export const main = async () => {
  const command = new QueryCommand({
    TableName: "CoffeeCrop",
    KeyConditionExpression:
      "OriginCountry = :originCountry AND RoastDate > :roastDate",
    ExpressionAttributeValues: {
      ":originCountry": "Ethiopia",
      ":roastDate": "2023-05-01",
    },
    ConsistentRead: true,
  });

  const response = await docClient.send(command);
  console.log(response);
  return response;
};
```

- For more information, see AWS SDK for JavaScript Developer Guide.
- For API details, see Query in *AWS SDK for JavaScript API Reference*.

SDK for JavaScript (v2)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create DynamoDB document client
var docClient = new AWS.DynamoDB.DocumentClient({ apiVersion: "2012-08-10" });

var params = {
  ExpressionAttributeValues: {
    ":s": 2,
    ":e": 9,
    ":topic": "PHRASE",
  },
  KeyConditionExpression: "Season = :s and Episode > :e",
  FilterExpression: "contains (Subtitle, :topic)",
  TableName: "EPISODES_TABLE",
};

docClient.query(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data.Items);
  }
});
```

- For more information, see AWS SDK for JavaScript Developer Guide.
- For API details, see Query in *AWS SDK for JavaScript API Reference*.

Kotlin

SDK for Kotlin

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
suspend fun queryDynTable(
    tableNameVal: String,
    partitionKeyName: String,
    partitionKeyVal: String,
    partitionAlias: String,
): Int {
    val attrNameAlias = mutableMapOf<String, String>()
    attrNameAlias[partitionAlias] = partitionKeyName

    // Set up mapping of the partition name with the value.
    val attrValues = mutableMapOf<String, AttributeValue>()
    attrValues[":$partitionKeyName"] = AttributeValue.S(partitionKeyVal)

    val request =
        QueryRequest {
            tableName = tableNameVal
            keyConditionExpression = "$partitionAlias = :$partitionKeyName"
            expressionAttributeNames = attrNameAlias
            this.expressionAttributeValues = attrValues
        }

    DynamoDbClient.fromEnvironment { region = "us-east-1" }.use { ddb ->
        val response = ddb.query(request)
        return response.count
    }
}
```

- For API details, see Query in *AWS SDK for Kotlin API reference*.

PHP

SDK for PHP

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
        $birthKey = [
            'Key' => [
                'year' => [
                    'N' => "$birthYear",
                ],
            ],
        ];
        $result = $service->query($tableName, $birthKey);

    public function query(string $tableName, $key)
    {
        $expressionAttributeValues = [];
        $expressionAttributeNames = [];
        $keyConditionExpression = "";
        $index = 1;
        foreach ($key as $name => $value) {
            $keyConditionExpression .= "#" . array_key_first($value) . " = :v
$index,";
            $expressionAttributeNames["#" . array_key_first($value)] =
 array_key_first($value);
            $hold = array_pop($value);
            $expressionAttributeValues[":v$index"] = [
                array_key_first($hold) => array_pop($hold),
            ];
        }
        $keyConditionExpression = substr($keyConditionExpression, 0, -1);
        $query = [
            'ExpressionAttributeValues' => $expressionAttributeValues,
            'ExpressionAttributeNames' => $expressionAttributeNames,
            'KeyConditionExpression' => $keyConditionExpression,
            'TableName' => $tableName,
        ];
```

```
        return $this->dynamoDbClient->query($query);
    }
```

- For API details, see Query in *AWS SDK for PHP API Reference*.

PowerShell

Tools for PowerShell V4

Example 1: Invokes a query that returns DynamoDB items with the specified SongTitle and Artist.

```
$invokeDDBQuery = @{
    TableName = 'Music'
    KeyConditionExpression = ' SongTitle = :SongTitle and Artist = :Artist'
    ExpressionAttributeValues = @{
        ':SongTitle' = 'Somewhere Down The Road'
        ':Artist' = 'No One You Know'
    } | ConvertTo-DDBItem
}
Invoke-DDBQuery @invokeDDBQuery | ConvertFrom-DDBItem
```

Output:

```
Name                           Value
----                           -----
Genre                          Country
Artist                         No One You Know
Price                          1.94
CriticRating                   9
SongTitle                      Somewhere Down The Road
AlbumTitle                     Somewhat Famous
```

- For API details, see Query in *AWS Tools for PowerShell Cmdlet Reference (V4)*.

Tools for PowerShell V5

Example 1: Invokes a query that returns DynamoDB items with the specified SongTitle and Artist.

```
$invokeDDBQuery = @{
```

```
    TableName = 'Music'
    KeyConditionExpression = ' SongTitle = :SongTitle and Artist = :Artist'
    ExpressionAttributeValues = @{
        ':SongTitle' = 'Somewhere Down The Road'
        ':Artist' = 'No One You Know'
    } | ConvertTo-DDBItem
}
Invoke-DDBQuery @invokeDDBQuery | ConvertFrom-DDBItem
```

Output:

```
Name                           Value
----                           -----
Genre                          Country
Artist                         No One You Know
Price                          1.94
CriticRating                   9
SongTitle                      Somewhere Down The Road
AlbumTitle                     Somewhat Famous
```

- For API details, see Query in *AWS Tools for PowerShell Cmdlet Reference (V5)*.

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Query items by using a key condition expression.

```
class Movies:
    """Encapsulates an Amazon DynamoDB table of movie data.

    Example data structure for a movie record in this table:
        {
            "year": 1999,
            "title": "For Love of the Game",
```

```
            "info": {
                "directors": ["Sam Raimi"],
                "release_date": "1999-09-15T00:00:00Z",
                "rating": 6.3,
                "plot": "A washed up pitcher flashes through his career.",
                "rank": 4987,
                "running_time_secs": 8220,
                "actors": [
                    "Kevin Costner",
                    "Kelly Preston",
                    "John C. Reilly"
                ]
            }
        }
    """

    def __init__(self, dyn_resource):
        """
        :param dyn_resource: A Boto3 DynamoDB resource.
        """
        self.dyn_resource = dyn_resource
        # The table variable is set during the scenario in the call to
        # 'exists' if the table exists. Otherwise, it is set by 'create_table'.
        self.table = None


    def query_movies(self, year):
        """
        Queries for movies that were released in the specified year.

        :param year: The year to query.
        :return: The list of movies that were released in the specified year.
        """
        try:
            response =
self.table.query(KeyConditionExpression=Key("year").eq(year))
        except ClientError as err:
            logger.error(
                "Couldn't query for movies released in %s. Here's why: %s: %s",
                year,
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
```

```
        else:
            return response["Items"]
```

Query items and project them to return a subset of data.

```
class UpdateQueryWrapper:
    def __init__(self, table):
        self.table = table


    def query_and_project_movies(self, year, title_bounds):
        """
        Query for movies that were released in a specified year and that have
 titles
        that start within a range of letters. A projection expression is used
        to return a subset of data for each movie.

        :param year: The release year to query.
        :param title_bounds: The range of starting letters to query.
        :return: The list of movies.
        """
        try:
            response = self.table.query(
                ProjectionExpression="#yr, title, info.genres, info.actors[0]",
                ExpressionAttributeNames={"#yr": "year"},
                KeyConditionExpression=(
                    Key("year").eq(year)
                    & Key("title").between(
                        title_bounds["first"], title_bounds["second"]
                    )
                ),
            )
        except ClientError as err:
            if err.response["Error"]["Code"] == "ValidationException":
                logger.warning(
                    "There's a validation error. Here's the message: %s: %s",
                    err.response["Error"]["Code"],
                    err.response["Error"]["Message"],
                )
            else:
                logger.error(
```

```
                    "Couldn't query for movies. Here's why: %s: %s",
                    err.response["Error"]["Code"],
                    err.response["Error"]["Message"],
                )
                raise
        else:
            return response["Items"]
```

- For API details, see Query in *AWS SDK for Python (Boto3) API Reference*.

Ruby

SDK for Ruby

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
class DynamoDBBasics
  attr_reader :dynamo_resource, :table

  def initialize(table_name)
    client = Aws::DynamoDB::Client.new(region: 'us-east-1')
    @dynamo_resource = Aws::DynamoDB::Resource.new(client: client)
    @table = @dynamo_resource.table(table_name)
  end

  # Queries for movies that were released in the specified year.
  #
  # @param year [Integer] The year to query.
  # @return [Array] The list of movies that were released in the specified year.
  def query_items(year)
    response = @table.query(
      key_condition_expression: '#yr = :year',
      expression_attribute_names: { '#yr' => 'year' },
      expression_attribute_values: { ':year' => year }
    )
  rescue Aws::DynamoDB::Errors::ServiceError => e
```

```
    puts("Couldn't query for movies released in #{year}. Here's why:")
    puts("\t#{e.code}: #{e.message}")
    raise
  else
    response.items
  end
```

- For API details, see Query in *AWS SDK for Ruby API Reference*.

Rust

SDK for Rust

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Find the movies made in the specified year.

```
pub async fn movies_in_year(
    client: &Client,
    table_name: &str,
    year: u16,
) -> Result<Vec<Movie>, MovieError> {
    let results = client
        .query()
        .table_name(table_name)
        .key_condition_expression("#yr = :yyyy")
        .expression_attribute_names("#yr", "year")
        .expression_attribute_values(":yyyy",
 AttributeValue::N(year.to_string()))
        .send()
        .await?;

    if let Some(items) = results.items {
        let movies = items.iter().map(|v| v.into()).collect();
        Ok(movies)
    } else {
        Ok(vec![])
```

```
    }
}
```

- For API details, see Query in *AWS SDK for Rust API reference*.

SAP ABAP

SDK for SAP ABAP

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    TRY.
        " Query movies for a given year .
        DATA(lt_attributelist) = VALUE /aws1/
cl_dynattributevalue=>tt_attributevaluelist(
            ( NEW /aws1/cl_dynattributevalue( iv_n = |{ iv_year }| ) ) ).
        DATA(lt_key_conditions) = VALUE /aws1/cl_dyncondition=>tt_keyconditions(
          ( VALUE /aws1/cl_dyncondition=>ts_keyconditions_maprow(
          key = 'year'
          value = NEW /aws1/cl_dyncondition(
          it_attributevaluelist = lt_attributelist
          iv_comparisonoperator = |EQ|
          ) ) ) ).
        oo_result = lo_dyn->query(
          iv_tablename = iv_table_name
          it_keyconditions = lt_key_conditions ).
        DATA(lt_items) = oo_result->get_items( ).
        "You can loop over the results to get item attributes.
        LOOP AT lt_items INTO DATA(lt_item).
          DATA(lo_title) = lt_item[ key = 'title' ]-value.
          DATA(lo_year) = lt_item[ key = 'year' ]-value.
        ENDLOOP.
        DATA(lv_count) = oo_result->get_count( ).
        MESSAGE 'Item count is: ' && lv_count TYPE 'I'.
      CATCH /aws1/cx_dynresourcenotfoundex.
        MESSAGE 'The table or index does not exist' TYPE 'E'.
```

```
    ENDTRY.
```

- For API details, see Query in *AWS SDK for SAP ABAP API reference*.

Swift

SDK for Swift

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import AWSDynamoDB


    /// Get all the movies released in the specified year.
    ///
    /// - Parameter year: The release year of the movies to return.
    ///
    /// - Returns: An array of `Movie` objects describing each matching movie.
    ///
    func getMovies(fromYear year: Int) async throws -> [Movie] {
        do {
            guard let client = self.ddbClient else {
                throw MoviesError.UninitializedClient
            }

            let input = QueryInput(
                expressionAttributeNames: [
                    "#y": "year"
                ],
                expressionAttributeValues: [
                    ":y": .n(String(year))
                ],
                keyConditionExpression: "#y = :y",
                tableName: self.tableName
            )
            // Use "Paginated" to get all the movies.
```

```
            // This lets the SDK handle the 'lastEvaluatedKey' property in
"QueryOutput".

            let pages = client.queryPaginated(input: input)

            var movieList: [Movie] = []
            for try await page in pages {
                guard let items = page.items else {
                    print("Error: no items returned.")
                    continue
                }

                // Convert the found movies into `Movie` objects and return an
array
                // of them.

                for item in items {
                    let movie = try Movie(withItem: item)
                    movieList.append(movie)
                }
            }
            return movieList
        } catch {
            print("ERROR: getMovies:", dump(error))
            throw error
        }
    }
```

- For API details, see Query in *AWS SDK for Swift API reference*.

For more DynamoDB examples, see Code examples for DynamoDB using AWS SDKs.

To create a global secondary index for your table, proceed to Step 6: (Optional) Delete your DynamoDB table to clean up resources.

Step 6: (Optional) Delete your DynamoDB table to clean up resources

If you no longer need the Amazon DynamoDB table that you created for the tutorial, you can delete it. This step helps ensure that you aren't charged for resources that you aren't using. You can use the DynamoDB console or the AWS CLI to delete the `Music` table that you created in Step 1: Create a table in DynamoDB.

For more information about table operations in DynamoDB, see Working with tables and data in DynamoDB.

AWS Management Console

To delete the `Music` table using the console:

1. Open the DynamoDB console at https://console.aws.amazon.com/dynamodb/.
2. In the left navigation pane, choose **Tables**.
3. Choose the checkbox beside the **Music** table in the table list.
4. Choose **Delete**.

AWS CLI

The following AWS CLI example deletes the `Music` table using `delete-table`.

```
aws dynamodb delete-table --table-name Music
```

AWS SDK

The following code examples show how to delete a DynamoDB table using an AWS SDK.

.NET

SDK for .NET (v4)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    /// <summary>
    /// Deletes a DynamoDB table.
    /// </summary>
    /// <param name="tableName">The name of the table to delete.</param>
    /// <returns>A Boolean value indicating the success of the operation.</
returns>
    public async Task<bool> DeleteTableAsync(string tableName)
    {
        try
        {
            var request = new DeleteTableRequest
            {
                TableName = tableName,
            };

            var response = await _amazonDynamoDB.DeleteTableAsync(request);

            Console.WriteLine($"Table {response.TableDescription.TableName}
 successfully deleted.");
            return true;

        }
        catch (ResourceNotFoundException ex)
        {
            Console.WriteLine($"Table {tableName} was not found and cannot be
 deleted. {ex.Message}");
            return false;
        }
        catch (AmazonDynamoDBException ex)
        {
            Console.WriteLine($"An Amazon DynamoDB error occurred while deleting
 table {tableName}. {ex.Message}");
            return false;
        }
        catch (Exception ex)
        {
            Console.WriteLine($"An error occurred while deleting table
 {tableName}. {ex.Message}");
            return false;
        }
    }
```

- For API details, see DeleteTable in *AWS SDK for .NET API Reference*.

Bash

AWS CLI with Bash script

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
###############################################################################
# function dynamodb_delete_table
#
# This function deletes a DynamoDB table.
#
# Parameters:
#       -n table_name  -- The name of the table to delete.
#
#  Returns:
#       0 - If successful.
#       1 - If it fails.
###############################################################################
function dynamodb_delete_table() {
  local table_name response
  local option OPTARG # Required to use getopts command in a function.

  # bashsupport disable=BP5008
  function usage() {
    echo "function dynamodb_delete_table"
    echo "Deletes an Amazon DynamoDB table."
    echo " -n table_name  -- The name of the table to delete."
    echo ""
  }

  # Retrieve the calling parameters.
  while getopts "n:h" option; do
    case "${option}" in
      n) table_name="${OPTARG}" ;;
      h)
        usage
```

```
        return 0
        ;;
      \?)
        echo "Invalid parameter"
        usage
        return 1
        ;;
    esac
  done
  export OPTIND=1

  if [[ -z "$table_name" ]]; then
    errecho "ERROR: You must provide a table name with the -n parameter."
    usage
    return 1
  fi

  iecho "Parameters:\n"
  iecho "    table_name:   $table_name"
  iecho ""

  response=$(aws dynamodb delete-table \
    --table-name "$table_name")

  local error_code=${?}

  if [[ $error_code -ne 0 ]]; then
    aws_cli_error_log $error_code
    errecho "ERROR: AWS reports delete-table operation failed.$response"
    return 1
  fi

  return 0
}
```

The utility functions used in this example.

```
###############################################################################
# function iecho
#
# This function enables the script to display the specified text only if
# the global variable $VERBOSE is set to true.
```

```
#####################################################################################
function iecho() {
  if [[ $VERBOSE == true ]]; then
    echo "$@"
  fi
}

#####################################################################################
# function errecho
#
# This function outputs everything sent to it to STDERR (standard error output).
#####################################################################################
function errecho() {
  printf "%s\n" "$*" 1>&2
}

#####################################################################################
# function aws_cli_error_log()
#
# This function is used to log the error messages from the AWS CLI.
#
# See https://docs.aws.amazon.com/cli/latest/topic/return-codes.html#cli-aws-
help-return-codes.
#
# The function expects the following argument:
#         $1 - The error code returned by the AWS CLI.
#
#  Returns:
#          0: - Success.
#
#####################################################################################
function aws_cli_error_log() {
  local err_code=$1
  errecho "Error code : $err_code"
  if [ "$err_code" == 1 ]; then
    errecho "  One or more S3 transfers failed."
  elif [ "$err_code" == 2 ]; then
    errecho "  Command line failed to parse."
  elif [ "$err_code" == 130 ]; then
    errecho "  Process received SIGINT."
  elif [ "$err_code" == 252 ]; then
    errecho "  Command syntax invalid."
  elif [ "$err_code" == 253 ]; then
    errecho "  The system environment or configuration was invalid."
```

```
  elif [ "$err_code" == 254 ]; then
    errecho "  The service returned an error."
  elif [ "$err_code" == 255 ]; then
    errecho "  255 is a catch-all error."
  fi

  return 0
}
```

- For API details, see DeleteTable in *AWS CLI Command Reference*.

C++

SDK for C++

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
//! Delete an Amazon DynamoDB table.
/*!
  \sa deleteTable()
  \param tableName: The DynamoDB table name.
  \param clientConfiguration: AWS client configuration.
  \return bool: Function succeeded.
*/
bool AwsDoc::DynamoDB::deleteTable(const Aws::String &tableName,
                                   const Aws::Client::ClientConfiguration
 &clientConfiguration) {
    Aws::DynamoDB::DynamoDBClient dynamoClient(clientConfiguration);

    Aws::DynamoDB::Model::DeleteTableRequest request;
    request.SetTableName(tableName);

    const Aws::DynamoDB::Model::DeleteTableOutcome &result =
 dynamoClient.DeleteTable(
            request);
    if (result.IsSuccess()) {
        std::cout << "Your table \""
```

```
                  << result.GetResult().GetTableDescription().GetTableName()
                  << " was deleted.\n";
    }
    else {
        std::cerr << "Failed to delete table: " << result.GetError().GetMessage()
                  << std::endl;
    }

    return result.IsSuccess();
}
```

- For API details, see DeleteTable in *AWS SDK for C++ API Reference*.

CLI

AWS CLI

To delete a table

The following `delete-table` example deletes the `MusicCollection` table.

```
aws dynamodb delete-table \
    --table-name MusicCollection
```

Output:

```
{
    "TableDescription": {
        "TableStatus": "DELETING",
        "TableSizeBytes": 0,
        "ItemCount": 0,
        "TableName": "MusicCollection",
        "ProvisionedThroughput": {
            "NumberOfDecreasesToday": 0,
            "WriteCapacityUnits": 5,
            "ReadCapacityUnits": 5
        }
    }
}
```

For more information, see Deleting a Table in the *Amazon DynamoDB Developer Guide*.

- For API details, see DeleteTable in *AWS CLI Command Reference*.

Go

SDK for Go V2

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import (
 "context"
 "errors"
 "log"
 "time"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/attributevalue"
 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/expression"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb/types"
)

// TableBasics encapsulates the Amazon DynamoDB service actions used in the
 examples.
// It contains a DynamoDB service client that is used to act on the specified
 table.
type TableBasics struct {
 DynamoDbClient *dynamodb.Client
 TableName      string
}



// DeleteTable deletes the DynamoDB table and all of its data.
func (basics TableBasics) DeleteTable(ctx context.Context) error {
 _, err := basics.DynamoDbClient.DeleteTable(ctx, &dynamodb.DeleteTableInput{
  TableName: aws.String(basics.TableName)})
 if err != nil {
```

```
  log.Printf("Couldn't delete table %v. Here's why: %v\n", basics.TableName, err)
 }
 return err
}
```

- For API details, see DeleteTable in *AWS SDK for Go API Reference*.

Java

SDK for Java 2.x

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.DeleteTableRequest;

/**
 * Before running this Java V2 code example, set up your development
 * environment, including your credentials.
 *
 * For more information, see the following documentation topic:
 *
 * https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-
started.html
 */

public class DeleteTable {
    public static void main(String[] args) {
        final String usage = """

                Usage:
                    <tableName>

                Where:
```

```
                    tableName - The Amazon DynamoDB table to delete (for example,
 Music3).

                **Warning** This program will delete the table that you specify!
                """;

        if (args.length != 1) {
            System.out.println(usage);
            System.exit(1);
        }

        String tableName = args[0];
        System.out.format("Deleting the Amazon DynamoDB table %s...\n",
 tableName);
        Region region = Region.US_EAST_1;
        DynamoDbClient ddb = DynamoDbClient.builder()
                .region(region)
                .build();

        deleteDynamoDBTable(ddb, tableName);
        ddb.close();
    }

    public static void deleteDynamoDBTable(DynamoDbClient ddb, String tableName)
 {
        DeleteTableRequest request = DeleteTableRequest.builder()
                .tableName(tableName)
                .build();

        try {
            ddb.deleteTable(request);

        } catch (DynamoDbException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
        System.out.println(tableName + " was successfully deleted!");
    }
}
```

- For API details, see DeleteTable in *AWS SDK for Java 2.x API Reference*.

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import { DeleteTableCommand, DynamoDBClient } from "@aws-sdk/client-dynamodb";

const client = new DynamoDBClient({});

export const main = async () => {
  const command = new DeleteTableCommand({
    TableName: "DecafCoffees",
  });

  const response = await client.send(command);
  console.log(response);
  return response;
};
```

- For API details, see DeleteTable in *AWS SDK for JavaScript API Reference*.

SDK for JavaScript (v2)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create the DynamoDB service object
```

```
var ddb = new AWS.DynamoDB({ apiVersion: "2012-08-10" });

var params = {
  TableName: process.argv[2],
};

// Call DynamoDB to delete the specified table
ddb.deleteTable(params, function (err, data) {
  if (err && err.code === "ResourceNotFoundException") {
    console.log("Error: Table not found");
  } else if (err && err.code === "ResourceInUseException") {
    console.log("Error: Table in use");
  } else {
    console.log("Success", data);
  }
});
```

- For more information, see AWS SDK for JavaScript Developer Guide.
- For API details, see DeleteTable in *AWS SDK for JavaScript API Reference*.

Kotlin

SDK for Kotlin

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
suspend fun deleteDynamoDBTable(tableNameVal: String) {
    val request =
        DeleteTableRequest {
            tableName = tableNameVal
        }

    DynamoDbClient.fromEnvironment { region = "us-east-1" }.use { ddb ->
        ddb.deleteTable(request)
        println("$tableNameVal was deleted")
    }
```

```
}
```

- For API details, see DeleteTable in *AWS SDK for Kotlin API reference*.

PHP

SDK for PHP

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    public function deleteTable(string $TableName)
    {
        $this->customWaiter(function () use ($TableName) {
            return $this->dynamoDbClient->deleteTable([
                'TableName' => $TableName,
            ]);
        });
    }
```

- For API details, see DeleteTable in *AWS SDK for PHP API Reference*.

PowerShell

Tools for PowerShell V4

Example 1: Deletes the specified table. You are prompted for confirmation before the operation proceeds.

```
Remove-DDBTable -TableName "myTable"
```

Example 2: Deletes the specified table. You are not prompted for confirmation before the operation proceeds.

```
Remove-DDBTable -TableName "myTable" -Force
```

- For API details, see DeleteTable in *AWS Tools for PowerShell Cmdlet Reference (V4)*.

Tools for PowerShell V5

Example 1: Deletes the specified table. You are prompted for confirmation before the operation proceeds.

```
Remove-DDBTable -TableName "myTable"
```

Example 2: Deletes the specified table. You are not prompted for confirmation before the operation proceeds.

```
Remove-DDBTable -TableName "myTable" -Force
```

- For API details, see DeleteTable in *AWS Tools for PowerShell Cmdlet Reference (V5)*.

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
class Movies:
    """Encapsulates an Amazon DynamoDB table of movie data.

    Example data structure for a movie record in this table:
        {
            "year": 1999,
            "title": "For Love of the Game",
            "info": {
                "directors": ["Sam Raimi"],
                "release_date": "1999-09-15T00:00:00Z",
                "rating": 6.3,
```

```
                "plot": "A washed up pitcher flashes through his career.",
                "rank": 4987,
                "running_time_secs": 8220,
                "actors": [
                    "Kevin Costner",
                    "Kelly Preston",
                    "John C. Reilly"
                ]
            }
        }
    """

    def __init__(self, dyn_resource):
        """
        :param dyn_resource: A Boto3 DynamoDB resource.
        """
        self.dyn_resource = dyn_resource
        # The table variable is set during the scenario in the call to
        # 'exists' if the table exists. Otherwise, it is set by 'create_table'.
        self.table = None


    def delete_table(self):
        """
        Deletes the table.
        """
        try:
            self.table.delete()
            self.table = None
        except ClientError as err:
            logger.error(
                "Couldn't delete table. Here's why: %s: %s",
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
```

- For API details, see DeleteTable in *AWS SDK for Python (Boto3) API Reference*.

Ruby

SDK for Ruby

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
# Encapsulates an Amazon DynamoDB table of movie data.
class Scaffold
  attr_reader :dynamo_resource, :table_name, :table

  def initialize(table_name)
    client = Aws::DynamoDB::Client.new(region: 'us-east-1')
    @dynamo_resource = Aws::DynamoDB::Resource.new(client: client)
    @table_name = table_name
    @table = nil
    @logger = Logger.new($stdout)
    @logger.level = Logger::DEBUG
  end

  # Deletes the table.
  def delete_table
    @table.delete
    @table = nil
  rescue Aws::DynamoDB::Errors::ServiceError => e
    puts("Couldn't delete table. Here's why:")
    puts("\t#{e.code}: #{e.message}")
    raise
  end
```

- For API details, see DeleteTable in *AWS SDK for Ruby API Reference*.

Rust

SDK for Rust

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
pub async fn delete_table(client: &Client, table: &str) ->
 Result<DeleteTableOutput, Error> {
    let resp = client.delete_table().table_name(table).send().await;

    match resp {
        Ok(out) => {
            println!("Deleted table");
            Ok(out)
        }
        Err(e) => Err(Error::Unhandled(e.into())),
    }
}
```

- For API details, see DeleteTable in *AWS SDK for Rust API reference*.

SAP ABAP

SDK for SAP ABAP

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    TRY.
        lo_dyn->deletetable( iv_tablename = iv_table_name ).
        " Wait till the table is actually deleted.
        lo_dyn->get_waiter( )->tablenotexists(
```

```
          iv_max_wait_time = 200
          iv_tablename     = iv_table_name ).
        MESSAGE 'Table ' && iv_table_name && ' deleted.' TYPE 'I'.
      CATCH /aws1/cx_dynresourcenotfoundex.
        MESSAGE 'The table ' && iv_table_name && ' does not exist' TYPE 'E'.
      CATCH /aws1/cx_dynresourceinuseex.
        MESSAGE 'The table cannot be deleted since it is in use' TYPE 'E'.
    ENDTRY.
```

- For API details, see DeleteTable in *AWS SDK for SAP ABAP API reference*.

Swift

SDK for Swift

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import AWSDynamoDB


    ///
    /// Deletes the table from Amazon DynamoDB.
    ///
    func deleteTable() async throws {
        do {
            guard let client = self.ddbClient else {
                throw MoviesError.UninitializedClient
            }

            let input = DeleteTableInput(
                tableName: self.tableName
            )
            _ = try await client.deleteTable(input: input)
        } catch {
            print("ERROR: deleteTable:", dump(error))
            throw error
        }
```

```
    }
```

- For API details, see DeleteTable in *AWS SDK for Swift API reference*.

For more DynamoDB examples, see Code examples for DynamoDB using AWS SDKs.

Continue learning about DynamoDB

For more information about using Amazon DynamoDB, see the following topics:

- Working with tables and data in DynamoDB
- Working with items and attributes in DynamoDB
- Querying tables in DynamoDB
- Using Global Secondary Indexes in DynamoDB
- Working with transactions
- In-memory acceleration with DynamoDB Accelerator (DAX)
- Programming with DynamoDB and the AWS SDKs

Generate infrastructure code for Amazon DynamoDB using Console-to-Code

Amazon Q Developer's Console-to-Code feature simplifies infrastructure management for Amazon DynamoDB by transforming manual table creation steps into reproducible automation code. This capability helps developers efficiently scale database resource configuration across their environments. For more information, see Automating AWS services with Amazon Q Developer Console-to-Code.

Console-to-Code captures detailed DynamoDB table configurations, including partition keys, sort keys, provisioned throughput settings, and secondary indexes, and converts these into precise infrastructure-as-code templates. Using generative AI, the tool ensures generated code maintains the parameter compatibility established during your console workflow.

Developers can generate DynamoDB infrastructure code in multiple formats, such as:

- AWS Cloud Development Kit (AWS CDK) in TypeScript, Python, and Java
- CloudFormation in YAML or JSON

This approach allows teams to:

- Standardize database resource management
- Implement version-controlled infrastructure
- Reduce manual configuration errors

Console-to-Code for Amazon DynamoDB is available in all commercial AWS Regions, providing a powerful solution for transforming manual configurations into automated, reproducible infrastructure code.

How it works

When using Console-to-Code with DynamoDB, the process typically involves:

1. **Prototyping in the console** – Use the DynamoDB console to create and configure resources like tables. See Connect to Amazon DynamoDB for more information.
2. **Recording actions** – Console-to-Code records these actions as you perform them.
3. **Code generation** – The feature uses Amazon Q Developer's generative AI capabilities to transform your console actions into reusable code in your preferred format.
4. **Code customization** – You can then copy or download this code and further customize it for your production workloads.

Benefits of using Console-to-Code with DynamoDB

Simplified automation

Convert manual DynamoDB table creation and configuration into reusable code with a single click.

Best practices

Generated code follows AWS guided best practices for reliable deployments.

Bridge between console and code

You don't have to choose between using the AWS Management Console or Infrastructure-as-Code (IaC). You can use both approaches together.

Accelerated development

Get started quickly with automation code that can be further customized for production use.

Example use cases

- Creating DynamoDB tables with specific attributes, keys, and capacity settings
- Setting up Global Secondary Indexes and Local Secondary Indexes
- Configuring auto-scaling policies for DynamoDB tables
- Establishing backup and restore configurations
- Creating and managing DynamoDB Streams

Getting started

To start using Console-to-Code with DynamoDB:

1. Sign in to the AWS Management Console and open the DynamoDB console at https://console.aws.amazon.com/dynamodbv2/.
2. Begin creating or modifying DynamoDB resources through the console interface.
3. Use the Console-to-Code feature to generate code for your actions in your preferred format.
4. Copy or download the generated code and customize it as needed for your specific requirements.

For more information and detailed instructions on how to use Console-to-Code, see Automating AWS services with Amazon Q Developer Console-to-Code in the *Amazon Q Developer User Guide*.

Amazon DynamoDB: How it works

The following sections provide an overview of Amazon DynamoDB service components and how they interact.

Topics

- Cheat sheet for DynamoDB
- Core components of Amazon DynamoDB
- DynamoDB API
- Supported data types and naming rules in Amazon DynamoDB
- DynamoDB table classes
- Partitions and data distribution in DynamoDB
- Learn how to go from SQL to NoSQL
- Amazon DynamoDB learning resources and tools

Cheat sheet for DynamoDB

This cheat sheet provides a quick reference for working with Amazon DynamoDB and its various AWS SDKs.

Initial setup

1. Sign up for AWS.
2. Get an AWS access key to access DynamoDB programmatically.
3. Configure your DynamoDB credentials.

See also:

- Setting up DynamoDB (web service)
- Getting started with DynamoDB
- Basic overview of core components

SDK or CLI

Choose your preferred SDK, or set up the AWS CLI.

> **Note**
>
> When you use the AWS CLI on Windows, a backslash (\) that is not inside a quote is treated as a carriage return. Also, you must escape any quotes and braces inside other quotes. For an example, see the **Windows** tab in "Create a table" in the next section.

See also:

- AWS CLI with DynamoDB
- Getting started with DynamoDB - step 2

Basic actions

This section provides code for basic DynamoDB tasks. For more information about these tasks, see Getting started with DynamoDB and the AWS SDKs.

Create a table

Default

```
aws dynamodb create-table \
    --table-name Music \
    --attribute-definitions \
        AttributeName=Artist,AttributeType=S \
        AttributeName=SongTitle,AttributeType=S \
    --key-schema AttributeName=Artist,KeyType=HASH
 AttributeName=SongTitle,KeyType=RANGE \
    --billing-mode PAY_PER_REQUEST \
    --table-class STANDARD
```

Windows

```
aws dynamodb create-table ^
    --table-name Music ^
    --attribute-definitions ^
```

```
        AttributeName=Artist,AttributeType=S ^
        AttributeName=SongTitle,AttributeType=S ^
    --key-schema AttributeName=Artist,KeyType=HASH
AttributeName=SongTitle,KeyType=RANGE ^
    --billing-mode PAY_PER_REQUEST ^
    --table-class STANDARD
```

Write item to a table

```
aws dynamodb put-item \ --table-name Music \ --item file://item.json
```

Read item from a table

```
aws dynamodb get-item \ --table-name Music \ --item file://item.json
```

Delete item from a table

```
aws dynamodb delete-item --table-name Music --key file://key.json
```

Query a table

```
aws dynamodb query --table-name Music
--key-condition-expression "ArtistName=:Artist and SongName=:Songtitle"
```

Delete a table

```
aws dynamodb delete-table --table-name Music
```

List table names

```
aws dynamodb list-tables
```

Naming rules

- All names must be encoded using UTF-8 and are case sensitive.
- Table names and index names must be between 3 and 255 characters long, and can contain only the following characters:

- `a-z`
- `A-Z`
- `0-9`
- _(underscore)
- `-`(hyphen)
- `.`(dot)

- Attribute names must be at least one character long, and less than 64 KB in size.

For more information, see Naming rules.

Service quota basics

Read and write units

- **Read capacity unit (RCU)** – One strongly consistent read per second, or two eventually consistent reads per second, for items up to 4 KB in size.
- **Write capacity unit (WCU)** – One write per second, for items up to 1 KB in size.

Table limits

- **Table size** – There is no practical limit on table size. Tables are unconstrained in terms of the number of items or the number of bytes.
- **Number of tables** – For any AWS account, there is an initial quota of 2,500 tables per AWS Region.
- **Page size limit for query and scan** – There is a limit of 1 MB per page, per query or scan. If your query parameters or scan operation on a table result in more than 1 MB of data, DynamoDB returns the initial matching items. It also returns a `LastEvaluatedKey` property that you can use in a new request to read the next page.

Indexes

- **Local secondary indexes (LSIs)** – You can define a maximum of five local secondary indexes. LSIs are primarily useful when an index must have strong consistency with the base table.
- **Global secondary indexes (GSIs)** – There is a default quota of 20 global secondary indexes per table.

- **Projected secondary index attributes per table** – You can project a total of up to 100 attributes into all of a table's local and global secondary indexes. This only applies to user-specified projected attributes.

Partition keys

- The minimum length of a partition key value is 1 byte. The maximum length is 2048 bytes.
- There is no practical limit on the number of distinct partition key values, for tables or for secondary indexes.
- The minimum length of a sort key value is 1 byte. The maximum length is 1024 bytes.
- In general, there is no practical limit on the number of distinct sort key values per partition key value. The exception is for tables with secondary indexes.

For more information on secondary indexes, partition key design, and sort key design, see Best practices.

Limits for commonly used data types

- **String** – The length of a string is constrained by the maximum item size of 400 KB. Strings are Unicode with UTF-8 binary encoding.
- **Number** – A number can have up to 38 digits of precision, and can be positive, negative, or zero.
- **Binary** – The length of a binary is constrained by the maximum item size of 400 KB. Applications that work with binary attributes must encode the data in base64 encoding before sending it to DynamoDB.

For a full list of supported data types, see Data types. For more information, also see Service quotas.

Items, attributes, and expression parameters

The maximum item size in DynamoDB is 400 KB, which includes both attribute name binary length (UTF-8 length) and attribute value binary lengths (UTF-8 length). The attribute name counts towards the size limit.

There is no limit on the number of values in a list, map, or set, as long as the item that contains the values fits within the 400-KB item size limit.

For expression parameters, the maximum length of any expression string is 4 KB.

For more information about item size, attributes, and expression parameters, see Service quotas.

More information

- Security
- Monitoring and logging
- Working with streams
- Backups and Point-in-time recovery
- Integrating with other AWS services
- API reference
- Architecture Center: Database Best Practices
- Video tutorials
- DynamoDB forum

Core components of Amazon DynamoDB

In DynamoDB, tables, items, and attributes are the core components that you work with. A *table* is a collection of *items*, and each item is a collection of *attributes*. DynamoDB uses primary keys to uniquely identify each item in a table. You can use DynamoDB Streams to capture data modification events in DynamoDB tables.

There are limits in DynamoDB. For more information, see Quotas in Amazon DynamoDB.

The following video will give you an introductory look at tables, items, and attributes.

Tables, items, and attributes

Tables, items, and attributes

People

Primary key Partition key: PersonID	Attributes			
101	LastName	FirstName	Phone	
	Smith	Fred	555-4321	
102	LastName	FirstName	Address	
	Jones	Mary	{"Street":"123 Main","City":"Anytown","State":"OH","ZIPCode":12345}	
103	LastName	FirstName	FavoriteColor	Address
	Stephens	Howard	Blue	{"Street":"123 Main","City":"London","PostalCode":"ER3 5K8"}

The following are the basic DynamoDB components:

- **Tables** – Similar to other database systems, DynamoDB stores data in tables. A *table* is a collection of data. For example, see the example table called *People* that you could use to store personal contact information about friends, family, or anyone else of interest. You could also have a *Cars* table to store information about vehicles that people drive.
- **Items** – Each table contains zero or more items. An *item* is a group of attributes that is uniquely identifiable among all of the other items. In a *People* table, each item represents a person. For a *Cars* table, each item represents one vehicle. Items in DynamoDB are similar in many ways to rows, records, or tuples in other database systems. In DynamoDB, there is no limit to the number of items you can store in a table.
- **Attributes** – Each item is composed of one or more attributes. An *attribute* is a fundamental data element, something that does not need to be broken down any further. For example, an item in a *People* table contains attributes called *PersonID*, *LastName*, *FirstName*, and so on. For a *Department* table, an item might have attributes such as *DepartmentID*, *Name*, *Manager*, and so on. Attributes in DynamoDB are similar in many ways to fields or columns in other database systems.

The following diagram shows a table named *People* with some example items and attributes.

```
People

{
    "PersonID": 101,
    "LastName": "Smith",
    "FirstName": "Fred",
    "Phone": "555-4321"
}

{
    "PersonID": 102,
    "LastName": "Jones",
    "FirstName": "Mary",
    "Address": {
                "Street": "123 Main",
                "City": "Anytown",
                "State": "OH",
                "ZIPCode": 12345
    }
}

{
    "PersonID": 103,
    "LastName": "Stephens",
    "FirstName": "Howard",
    "Address": {
                "Street": "123 Main",
                "City": "London",
                "PostalCode": "ER3 5K8"
    },
    "FavoriteColor": "Blue"
}
```

Note the following about the *People* table:

- Each item in the table has a unique identifier, or primary key, that distinguishes the item from all of the others in the table. In the *People* table, the primary key consists of one attribute (*PersonID*).
- Other than the primary key, the *People* table is schemaless, which means that neither the attributes nor their data types need to be defined beforehand. Each item can have its own distinct attributes.

- Most of the attributes are *scalar*, which means that they can have only one value. Strings and numbers are common examples of scalars.
- Some of the items have a nested attribute (*Address*). DynamoDB supports nested attributes up to 32 levels deep.

The following is another example table named *Music* that you could use to keep track of your music collection.

```
Music

{
    "Artist": "No One You Know",
    "SongTitle": "My Dog Spot",
    "AlbumTitle": "Hey Now",
    "Price": 1.98,
    "Genre": "Country",
    "CriticRating": 8.4
}

{
    "Artist": "No One You Know",
    "SongTitle": "Somewhere Down The Road",
    "AlbumTitle": "Somewhat Famous",
    "Genre": "Country",
    "CriticRating": 8.4,
    "Year": 1984
}

{
    "Artist": "The Acme Band",
    "SongTitle": "Still in Love",
    "AlbumTitle": "The Buck Starts Here",
    "Price": 2.47,
    "Genre": "Rock",
    "PromotionInfo": {
        "RadioStationsPlaying": [
            "KHCR",
            "KQBX",
            "WTNR",
            "WJJH"
        ],
```

```
        "TourDates": {
            "Seattle": "20150622",
            "Cleveland": "20150630"
        },
        "Rotation": "Heavy"
    }
}

{
    "Artist": "The Acme Band",
    "SongTitle": "Look Out, World",
    "AlbumTitle": "The Buck Starts Here",
    "Price": 0.99,
    "Genre": "Rock"
}
```

Note the following about the *Music* table:

- The primary key for *Music* consists of two attributes (*Artist* and *SongTitle*). Each item in the table must have these two attributes. The combination of *Artist* and *SongTitle* distinguishes each item in the table from all of the others.
- Other than the primary key, the *Music* table is schemaless, which means that neither the attributes nor their data types need to be defined beforehand. Each item can have its own distinct attributes.
- One of the items has a nested attribute (*PromotionInfo*), which contains other nested attributes. DynamoDB supports nested attributes up to 32 levels deep.

For more information, see Working with tables and data in DynamoDB.

Primary key

When you create a table, in addition to the table name, you must specify the primary key of the table. The primary key uniquely identifies each item in the table, so that no two items can have the same key.

DynamoDB supports two different kinds of primary keys:

- **Partition key** – A simple primary key, composed of one attribute known as the *partition key*.

DynamoDB uses the partition key's value as input to an internal hash function. The output from the hash function determines the partition (physical storage internal to DynamoDB) in which the item will be stored.

In a table that has only a partition key, no two items can have the same partition key value.

The *People* table described in Tables, items, and attributes is an example of a table with a simple primary key (*PersonID*). You can access any item in the *People* table directly by providing the *PersonId* value for that item.

- **Partition key and sort key** – Referred to as a *composite primary key*, this type of key is composed of two attributes. The first attribute is the *partition key*, and the second attribute is the *sort key*.

  DynamoDB uses the partition key value as input to an internal hash function. The output from the hash function determines the partition (physical storage internal to DynamoDB) in which the item will be stored. All items with the same partition key value are stored together, in sorted order by sort key value.

  In a table that has a partition key and a sort key, it's possible for multiple items to have the same partition key value. However, those items must have different sort key values.

  The *Music* table described in Tables, items, and attributes is an example of a table with a composite primary key (*Artist* and *SongTitle*). You can access any item in the *Music* table directly, if you provide the *Artist* and *SongTitle* values for that item.

  A composite primary key gives you additional flexibility when querying data. For example, if you provide only the value for *Artist*, DynamoDB retrieves all of the songs by that artist. To retrieve only a subset of songs by a particular artist, you can provide a value for *Artist* along with a range of values for *SongTitle*.

> **Note**
>
> The partition key of an item is also known as its *hash attribute*. The term *hash attribute* derives from the use of an internal hash function in DynamoDB that evenly distributes data items across partitions, based on their partition key values.
> The sort key of an item is also known as its *range attribute*. The term *range attribute* derives from the way DynamoDB stores items with the same partition key physically close together, in sorted order by the sort key value.

Each primary key attribute must be a scalar (meaning that it can hold only a single value). The only data types allowed for primary key attributes are string, number, or binary. There are no such restrictions for other, non-key attributes.

Secondary indexes

You can create one or more secondary indexes on a table. A *secondary index* lets you query the data in the table using an alternate key, in addition to queries against the primary key. DynamoDB doesn't require that you use indexes, but they give your applications more flexibility when querying your data. After you create a secondary index on a table, you can read data from the index in much the same way as you do from the table.

DynamoDB supports two kinds of indexes:

- Global secondary index – An index with a partition key and sort key that can be different from those on the table. The primary key values in global secondary indexes don't need to be unique.
- Local secondary index – An index that has the same partition key as the table, but a different sort key.

In DynamoDB, global secondary indexes (GSIs) are indexes that span the entire table, allowing you to query across all partition keys. Local secondary indexes (LSIs) are indexes that have the same partition key as the base table but a different sort key.

Each table in DynamoDB has a quota of 20 global secondary indexes (default quota) and 5 local secondary indexes.

In the example *Music* table shown previously, you can query data items by *Artist* (partition key) or by *Artist* and *SongTitle* (partition key and sort key). What if you also wanted to query the data by *Genre* and *AlbumTitle*? To do this, you could create an index on *Genre* and *AlbumTitle*, and then query the index in much the same way as you'd query the *Music* table.

The following diagram shows the example *Music* table, with a new index called *GenreAlbumTitle*. In the index, *Genre* is the partition key and *AlbumTitle* is the sort key.

Music Table	***GenreAlbumTitle***
`{ "Artist": "No One You Know",`	`{ "Genre": "Country",`

Music Table	*GenreAlbumTitle*

<pre>{
    "SongTitle": "My Dog Spot",
    "AlbumTitle": "Hey Now",
    "Price": 1.98,
    "Genre": "Country",
    "CriticRating": 8.4
}</pre>
	<pre>{
	    "AlbumTitle": "Hey Now",
	    "Artist": "No One You Know",
	    "SongTitle": "My Dog Spot"
	}</pre>

<pre>{
    "Artist": "No One You Know",
    "SongTitle": "Somewhere Down The
 Road",
    "AlbumTitle": "Somewhat Famous",
    "Genre": "Country",
    "CriticRating": 8.4,
    "Year": 1984
}</pre>
	<pre>{
	    "Genre": "Country",
	    "AlbumTitle": "Somewhat Famous",
	    "Artist": "No One You Know",
	    "SongTitle": "Somewhere Down The
	 Road"
	}</pre>

Music Table	*GenreAlbumTitle*

<pre>{
    "Artist": "The Acme Band",
    "SongTitle": "Still in Love",
    "AlbumTitle": "The Buck Starts
 Here",
    "Price": 2.47,
    "Genre": "Rock",
    "PromotionInfo": {
        "RadioStationsPlaying": {
            "KHCR",
            "KQBX",
            "WTNR",
            "WJJH"
        },
        "TourDates": {
            "Seattle": "20150622",
            "Cleveland": "20150630"
        },
        "Rotation": "Heavy"
    }
}</pre>
	<pre>{
	    "Genre": "Rock",
	    "AlbumTitle": "The Buck Starts
	 Here",
	    "Artist": "The Acme Band",
	    "SongTitle": "Still In Love"
	}</pre>

<pre>{
    "Artist": "The Acme Band",
    "SongTitle": "Look Out, World",
    "AlbumTitle": "The Buck Starts
 Here",
    "Price": 0.99,
    "Genre": "Rock"
}</pre>
	<pre>{
	    "Genre": "Rock",
	    "AlbumTitle": "The Buck Starts
	 Here",
	    "Artist": "The Acme Band",
	    "SongTitle": "Look Out, World"
	}</pre>

Note the following about the *GenreAlbumTitle* index:

- Every index belongs to a table, which is called the *base table* for the index. In the preceding example, *Music* is the base table for the *GenreAlbumTitle* index.
- DynamoDB maintains indexes automatically. When you add, update, or delete an item in the base table, DynamoDB adds, updates, or deletes the corresponding item in any indexes that belong to that table.
- When you create an index, you specify which attributes will be copied, or *projected*, from the base table to the index. At a minimum, DynamoDB projects the key attributes from the base table into the index. This is the case with `GenreAlbumTitle`, where only the key attributes from the `Music` table are projected into the index.

You can query the *GenreAlbumTitle* index to find all albums of a particular genre (for example, all *Rock* albums). You can also query the index to find all albums within a particular genre that have certain album titles (for example, all *Country* albums with titles that start with the letter H).

For more information, see Improving data access with secondary indexes in DynamoDB.

DynamoDB Streams

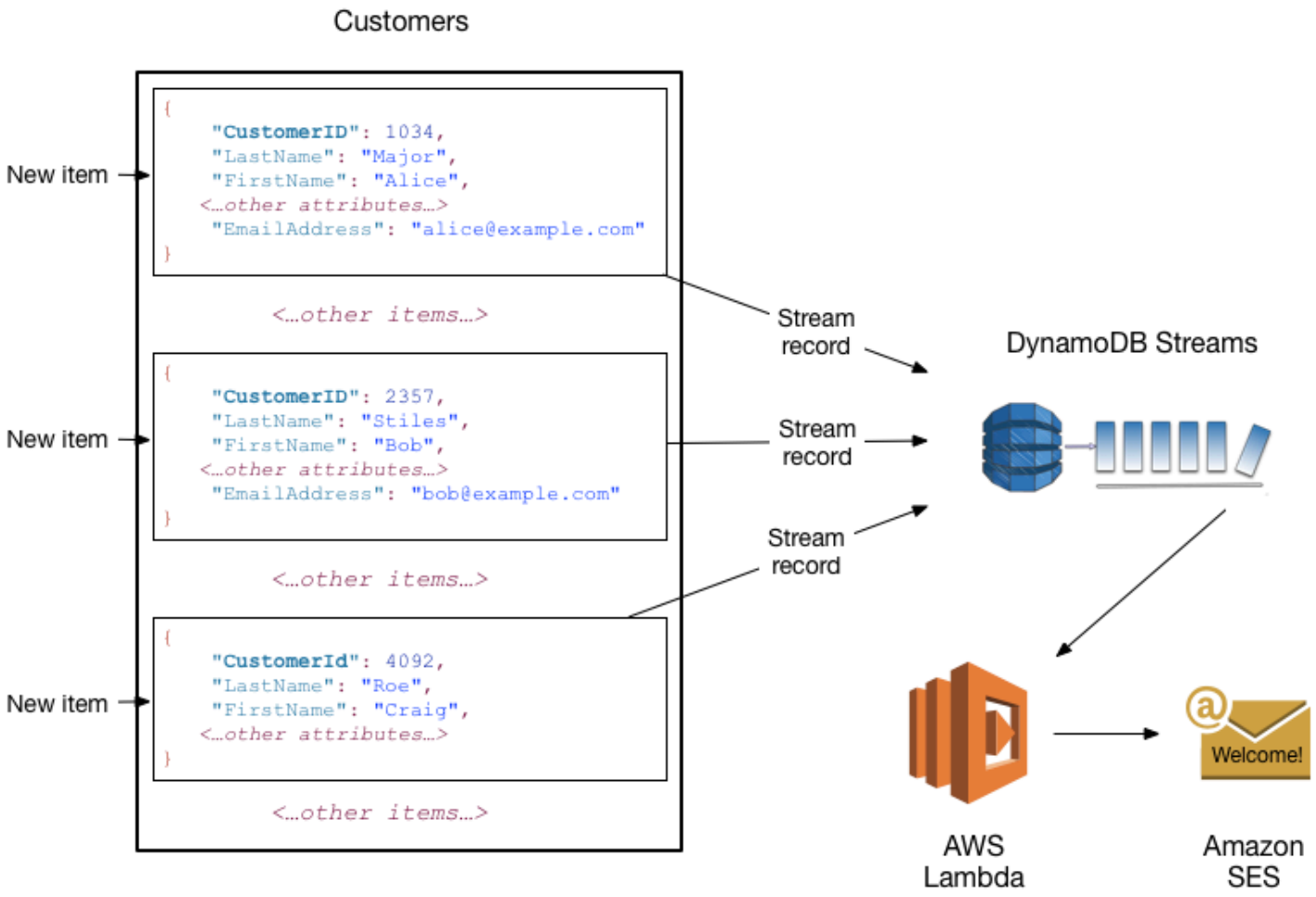
DynamoDB Streams is an optional feature that captures data modification events in DynamoDB tables. The data about these events appear in the stream in near-real time, and in the order that the events occurred.

Each event is represented by a *stream record*. If you enable a stream on a table, DynamoDB Streams writes a stream record whenever one of the following events occurs:

- A new item is added to the table: The stream captures an image of the entire item, including all of its attributes.
- An item is updated: The stream captures the "before" and "after" image of any attributes that were modified in the item.
- An item is deleted from the table: The stream captures an image of the entire item before it was deleted.

Each stream record also contains the name of the table, the event timestamp, and other metadata. Stream records have a lifetime of 24 hours; after that, they are automatically removed from the stream.

You can use DynamoDB Streams together with AWS Lambda to create a *trigger*—code that runs automatically whenever an event of interest appears in a stream. For example, consider a *Customers* table that contains customer information for a company. Suppose that you want to send a "welcome" email to each new customer. You could enable a stream on that table, and then associate the stream with a Lambda function. The Lambda function would run whenever a new stream record appears, but only process new items added to the *Customers* table. For any item that has an `EmailAddress` attribute, the Lambda function would invoke Amazon Simple Email Service (Amazon SES) to send an email to that address.



> **Note**
>
> In this example, the last customer, Craig Roe, will not receive an email because he doesn't have an `EmailAddress`.

In addition to triggers, DynamoDB Streams enables powerful solutions such as data replication within and across AWS Regions, materialized views of data in DynamoDB tables, data analysis using Kinesis materialized views, and much more.

For more information, see Change data capture for DynamoDB Streams.

DynamoDB API

To work with Amazon DynamoDB, your application must use a few simple API operations. The following is a summary of these operations, organized by category.

> **Note**
>
> For a full list of the API operations, see the Amazon DynamoDB API Reference.

Topics

- Control plane
- Data plane
- DynamoDB Streams
- Transactions

Control plane

Control plane operations let you create and manage DynamoDB tables. They also let you work with indexes, streams, and other objects that are dependent on tables.

- `CreateTable` – Creates a new table. Optionally, you can create one or more secondary indexes, and enable DynamoDB Streams for the table.
- `DescribeTable`– Returns information about a table, such as its primary key schema, throughput settings, and index information.
- `ListTables` – Returns the names of all of your tables in a list.
- `UpdateTable` – Modifies the settings of a table or its indexes, creates or removes new indexes on a table, or modifies DynamoDB Streams settings for a table.
- `DeleteTable` – Removes a table and all of its dependent objects from DynamoDB.

Data plane

Data plane operations let you perform create, read, update, and delete (also called *CRUD*) actions on data in a table. Some of the data plane operations also let you read data from a secondary index.

You can use PartiQL - a SQL-compatible query language for Amazon DynamoDB, to perform these CRUD operations or you can use DynamoDB's classic CRUD APIs that separates each operation into a distinct API call.

PartiQL - A SQL-compatible query language

- `ExecuteStatement` – Reads multiple items from a table. You can also write or update a single item from a table. When writing or updating a single item, you must specify the primary key attributes.
- `BatchExecuteStatement` – Writes, updates or reads multiple items from a table. This is more efficient than `ExecuteStatement` because your application only needs a single network round trip to write or read the items.

Classic APIs

Creating data

- `PutItem` – Writes a single item to a table. You must specify the primary key attributes, but you don't have to specify other attributes.
- `BatchWriteItem` – Writes up to 25 items to a table. This is more efficient than calling `PutItem` multiple times because your application only needs a single network round trip to write the items.

Reading data

- `GetItem` – Retrieves a single item from a table. You must specify the primary key for the item that you want. You can retrieve the entire item, or just a subset of its attributes.
- `BatchGetItem` – Retrieves up to 100 items from one or more tables. This is more efficient than calling `GetItem` multiple times because your application only needs a single network round trip to read the items.

- `Query` – Retrieves all items that have a specific partition key. You must specify the partition key value. You can retrieve entire items, or just a subset of their attributes. Optionally, you can apply a condition to the sort key values so that you only retrieve a subset of the data that has the same partition key. You can use this operation on a table, provided that the table has both a partition key and a sort key. You can also use this operation on an index, provided that the index has both a partition key and a sort key.
- `Scan` – Retrieves all items in the specified table or index. You can retrieve entire items, or just a subset of their attributes. Optionally, you can apply a filtering condition to return only the values that you are interested in and discard the rest.

Updating data

- `UpdateItem` – Modifies one or more attributes in an item. You must specify the primary key for the item that you want to modify. You can add new attributes and modify or remove existing attributes. You can also perform conditional updates, so that the update is only successful when a user-defined condition is met. Optionally, you can implement an atomic counter, which increments or decrements a numeric attribute without interfering with other write requests.

Deleting data

- `DeleteItem` – Deletes a single item from a table. You must specify the primary key for the item that you want to delete.
- `BatchWriteItem` – Deletes up to 25 items from one or more tables. This is more efficient than calling `DeleteItem` multiple times because your application only needs a single network round trip to delete the items.

> **Note**
>
> You can use `BatchWriteItem` to both create data and delete data.

DynamoDB Streams

DynamoDB Streams operations let you enable or disable a stream on a table, and allow access to the data modification records contained in a stream.

- `ListStreams` – Returns a list of all your streams, or just the stream for a specific table.

- `DescribeStream` – Returns information about a stream, such as its Amazon Resource Name (ARN) and where your application can begin reading the first few stream records.
- `GetShardIterator` – Returns a *shard iterator*, which is a data structure that your application uses to retrieve the records from the stream.
- `GetRecords` – Retrieves one or more stream records, using a given shard iterator.

Transactions

Transactions provide atomicity, consistency, isolation, and durability (ACID) enabling you to maintain data correctness in your applications more easily.

You can use PartiQL - a SQL-compatible query language for Amazon DynamoDB, to perform transactional operations or you can use DynamoDB's classic CRUD APIs that separates each operation into a distinct API call.

PartiQL - A SQL-compatible query language

- `ExecuteTransaction` – A batch operation that allows CRUD operations to multiple items both within and across tables with a guaranteed all-or-nothing result.

Classic APIs

- `TransactWriteItems` – A batch operation that allows `Put`, `Update`, and `Delete` operations to multiple items both within and across tables with a guaranteed all-or-nothing result.
- `TransactGetItems` – A batch operation that allows `Get` operations to retrieve multiple items from one or more tables.

Supported data types and naming rules in Amazon DynamoDB

This section describes the Amazon DynamoDB naming rules and the various data types that DynamoDB supports. There are limits that apply to data types. For more information, see Data types.

Topics

- Naming rules
- Data types

- Data type descriptors

Naming rules

Tables, attributes, and other objects in DynamoDB must have names. Names should be meaningful and concise—for example, names such as *Products*, *Books*, and *Authors* are self-explanatory.

The following are the naming rules for DynamoDB:

- All names must be encoded using UTF-8, and are case-sensitive.
- Table names and index names must be between 3 and 255 characters long, and can contain only the following characters:
  - `a-z`
  - `A-Z`
  - `0-9`
  - `_` (underscore)
  - `-` (dash)
  - `.` (dot)
- Attribute names must be at least one character long and less than 64 KB in size. It is considered best practice to keep your attribute names as short as possible. This helps reduce read request units consumed, as attribute names are included in metering of storage and throughput usage.

  The following are the exceptions. These attribute names must be no greater than 255 characters long:
  - Secondary index partition key names
  - Secondary index sort key names
  - The names of any user-specified projected attributes (applicable only to local secondary indexes)

Reserved words and special characters

DynamoDB has a list of reserved words and special characters. For a complete list, see Reserved words in DynamoDB. Also, the following characters have special meaning in DynamoDB: **#** (hash) and **:** (colon).

Although DynamoDB allows you to use these reserved words and special characters for names, we recommend that you avoid doing so because you have to define placeholder variables whenever you use these names in an expression. For more information, see Expression attribute names (aliases) in DynamoDB.

Data types

DynamoDB supports many different data types for attributes within a table. They can be categorized as follows:

- **Scalar Types** – A scalar type can represent exactly one value. The scalar types are number, string, binary, Boolean, and null.
- **Document Types** – A document type can represent a complex structure with nested attributes, such as what you would find in a JSON document. The document types are list and map.
- **Set Types** – A set type can represent multiple scalar values. The set types are string set, number set, and binary set.

When you create a table or a secondary index, you must specify the names and data types of each primary key attribute (partition key and sort key). Furthermore, each primary key attribute must be defined as type string, number, or binary.

DynamoDB is a NoSQL database and is *schemaless*. This means that other than the primary key attributes, you don't have to define any attributes or data types when you create tables. By comparison, relational databases require you to define the names and data types of each column when you create a table.

The following are descriptions of each data type, along with examples in JSON format.

Scalar types

The scalar types are number, string, binary, Boolean, and null.

Number

Numbers can be positive, negative, or zero. Numbers can have up to 38 digits of precision. Exceeding this results in an exception. If you need greater precision than 38 digits, you can use strings.

- Positive range: 1E-130 to 9.99999999999999999999999999999999999999E+125

- Negative range: -9.99999999999999999999999999999999999999E+125 to -1E-130

In DynamoDB, numbers are represented as variable length. Leading and trailing zeroes are trimmed.

All numbers are sent across the network to DynamoDB as strings to maximize compatibility across languages and libraries. However, DynamoDB treats them as number type attributes for mathematical operations.

You can use the number data type to represent a date or a timestamp. One way to do this is by using epoch time—the number of seconds since 00:00:00 UTC on 1 January 1970. For example, the epoch time `1437136300` represents 12:31:40 PM UTC on 17 July 2015.

For more information, see http://en.wikipedia.org/wiki/Unix_time.

String

Strings are Unicode with UTF-8 binary encoding. The minimum length of a string can be zero, if the attribute is not used as a key for an index or table, and is constrained by the maximum DynamoDB item size limit of 400 KB.

The following additional constraints apply to primary key attributes that are defined as type string:

- For a simple primary key, the maximum length of the first attribute value (the partition key) is 2048 bytes.
- For a composite primary key, the maximum length of the second attribute value (the sort key) is 1024 bytes.

DynamoDB collates and compares strings using the bytes of the underlying UTF-8 string encoding. For example, "a" (0x61) is greater than "A" (0x41), and "¿" (0xC2BF) is greater than "z" (0x7A).

You can use the string data type to represent a date or a timestamp. One way to do this is by using ISO 8601 strings, as shown in these examples:

- `2016-02-15`
- `2015-12-21T17:42:34Z`
- `20150311T122706Z`

For more information, see http://en.wikipedia.org/wiki/ISO_8601.

> **Note**
>
> Unlike conventional relational databases, DynamoDB does not natively support a date and time data type. It can be useful instead to store date and time data as a number data type, using Unix epoch time.

Binary

Binary type attributes can store any binary data, such as compressed text, encrypted data, or images. Whenever DynamoDB compares binary values, it treats each byte of the binary data as unsigned.

The length of a binary attribute can be zero, if the attribute is not used as a key for an index or table, and is constrained by the maximum DynamoDB item size limit of 400 KB.

If you define a primary key attribute as a binary type attribute, the following additional constraints apply:

- For a simple primary key, the maximum length of the first attribute value (the partition key) is 2048 bytes.
- For a composite primary key, the maximum length of the second attribute value (the sort key) is 1024 bytes.

Your applications must encode binary values in base64-encoded format before sending them to DynamoDB. Upon receipt of these values, DynamoDB decodes the data into an unsigned byte array and uses that as the length of the binary attribute.

The following example is a binary attribute, using base64-encoded text.

```
dGhpcyB0ZXh0IGlzIGJhc2U2NC1lbmNvZGVk
```

Boolean

A Boolean type attribute can store either `true` or `false`.

Null

Null represents an attribute with an unknown or undefined state.

Document types

The document types are list and map. These data types can be nested within each other, to represent complex data structures up to 32 levels deep.

There is no limit on the number of values in a list or a map, as long as the item containing the values fits within the DynamoDB item size limit (400 KB).

An attribute value can be an empty string or empty binary value if the attribute is not used for a table or index key. An attribute value cannot be an empty set (string set, number set, or binary set), however, empty lists and maps are allowed. Empty string and binary values are allowed within lists and maps. For more information, see Attributes.

List

A list type attribute can store an ordered collection of values. Lists are enclosed in square brackets: `[ ... ]`

A list is similar to a JSON array. There are no restrictions on the data types that can be stored in a list element, and the elements in a list element do not have to be of the same type.

The following example shows a list that contains two strings and a number.

```
FavoriteThings: ["Cookies", "Coffee", 3.14159]
```

> **Note**
>
> DynamoDB lets you work with individual elements within lists, even if those elements are deeply nested. For more information, see Using expressions in DynamoDB.

Map

A map type attribute can store an unordered collection of name-value pairs. Maps are enclosed in curly braces: `{ ... }`

A map is similar to a JSON object. There are no restrictions on the data types that can be stored in a map element, and the elements in a map do not have to be of the same type.

Maps are ideal for storing JSON documents in DynamoDB. The following example shows a map that contains a string, a number, and a nested list that contains another map.

```
{
    Day: "Monday",
    UnreadEmails: 42,
    ItemsOnMyDesk: [
        "Coffee Cup",
        "Telephone",
        {
            Pens: { Quantity : 3},
            Pencils: { Quantity : 2},
            Erasers: { Quantity : 1}
        }
    ]
}
```

Note

DynamoDB lets you work with individual elements within maps, even if those elements are deeply nested. For more information, see Using expressions in DynamoDB.

Sets

DynamoDB supports types that represent sets of number, string, or binary values. All the elements within a set must be of the same type. For example, a Number Set can only contain numbers and a String Set can only contain strings.

There is no limit on the number of values in a set, as long as the item containing the values fits within the DynamoDB item size limit (400 KB).

Each value within a set must be unique. The order of the values within a set is not preserved. Therefore, your applications must not rely on any particular order of elements within the set. DynamoDB does not support empty sets, however, empty string and binary values are allowed within a set.

The following example shows a string set, a number set, and a binary set:

```
["Black", "Green", "Red"]

[42.2, -19, 7.5, 3.14]
```

```
["U3Vubnk=", "UmFpbnk=", "U25vd3k="]
```

Data type descriptors

The low-level DynamoDB API protocol uses *Data type descriptors* as tokens that tell DynamoDB how to interpret each attribute.

The following is a complete list of DynamoDB data type descriptors:

- **S** – String
- **N** – Number
- **B** – Binary
- **BOOL** – Boolean
- **NULL** – Null
- **M** – Map
- **L** – List
- **SS** – String Set
- **NS** – Number Set
- **BS** – Binary Set

DynamoDB table classes

DynamoDB offers two table classes designed to help you optimize for cost. The DynamoDB Standard table class is the default, and is recommended for the vast majority of workloads. The DynamoDB Standard-Infrequent Access (DynamoDB Standard-IA) table class is optimized for tables where storage is the dominant cost. For example, tables that store infrequently accessed data, such as application logs, old social media posts, e-commerce order history, and past gaming achievements, are good candidates for the Standard-IA table class. See Amazon DynamoDB Pricing for pricing details.

Every DynamoDB table is associated with a table class (DynamoDB Standard by default). All secondary indexes associated with the table use the same table class. Each table class offers different pricing for data storage as well as for read and write requests. You can select the most cost-effective table class for your table based on its storage and throughput usage patterns.

The choice of a table class is not permanent—you can change this setting using the AWS Management Console, AWS CLI, or AWS SDK. DynamoDB also supports managing your table class using AWS CloudFormation for single-Region tables and global tables. To learn more about selecting your table class, see Considerations when choosing a table class in DynamoDB.

Partitions and data distribution in DynamoDB

Amazon DynamoDB stores data in partitions. A *partition* is an allocation of storage for a table, backed by solid state drives (SSDs) and automatically replicated across multiple Availability Zones within an AWS Region. Partition management is handled entirely by DynamoDB—you never have to manage partitions yourself.

When you create a table, the initial status of the table is `CREATING`. During this phase, DynamoDB allocates sufficient partitions to the table so that it can handle your provisioned throughput requirements. You can begin writing and reading table data after the table status changes to `ACTIVE`.

DynamoDB allocates additional partitions to a table in the following situations:

- If you increase the table's provisioned throughput settings beyond what the existing partitions can support.
- If an existing partition fills to capacity and more storage space is required.

Partition management occurs automatically in the background and is transparent to your applications. Your table remains available throughout and fully supports your provisioned throughput requirements.

For more details, see Partition key design.

Global secondary indexes in DynamoDB are also composed of partitions. The data in a global secondary index is stored separately from the data in its base table, but index partitions behave in much the same way as table partitions.

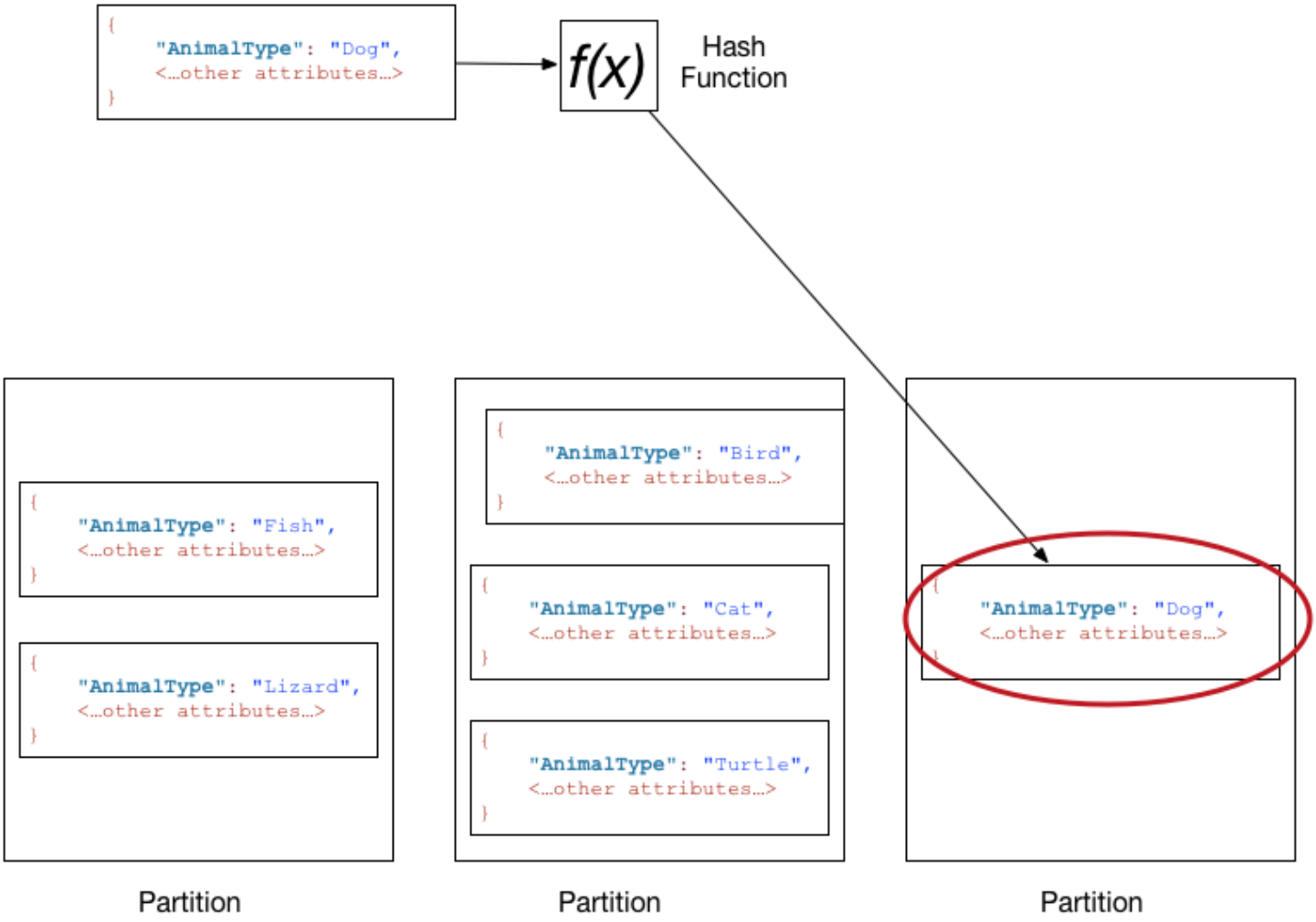
Data distribution: Partition key

If your table has a simple primary key (partition key only), DynamoDB stores and retrieves each item based on its partition key value.

To write an item to the table, DynamoDB uses the value of the partition key as input to an internal hash function. The output value from the hash function determines the partition in which the item will be stored.

To read an item from the table, you must specify the partition key value for the item. DynamoDB uses this value as input to its hash function, yielding the partition in which the item can be found.

The following diagram shows a table named *Pets*, which spans multiple partitions. The table's primary key is *AnimalType* (only this key attribute is shown). DynamoDB uses its hash function to determine where to store a new item, in this case based on the hash value of the string *Dog*. Note that the items are not stored in sorted order. Each item's location is determined by the hash value of its partition key.



> **Note**
>
> DynamoDB is optimized for uniform distribution of items across a table's partitions, no matter how many partitions there may be. We recommend that you choose a partition key that can have a large number of distinct values relative to the number of items in the table.

Data distribution: Partition key and sort key

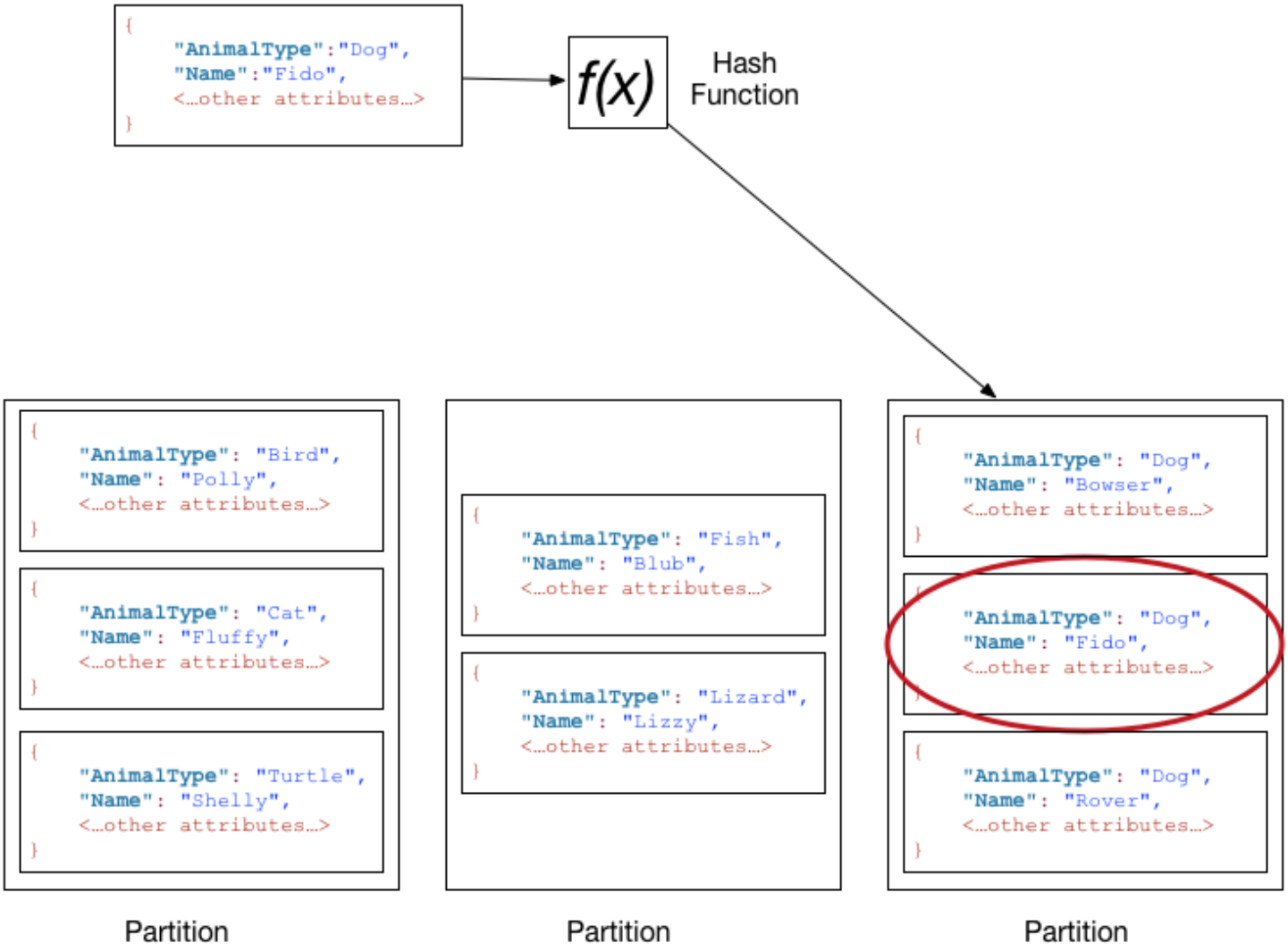
If the table has a composite primary key (partition key and sort key), DynamoDB calculates the hash value of the partition key in the same way as described in Data distribution: Partition key. However, it tends to keep items which have the same value of partition key close together and in sorted order by the sort key attribute's value. The set of items which have the same value of partition key is called an item collection. Item collections are optimized for efficient retrieval of ranges of the items within the collection. If your table doesn't have local secondary indexes, DynamoDB will automatically split your item collection over as many partitions as required to store the data and to serve read and write throughput.

To write an item to the table, DynamoDB calculates the hash value of the partition key to determine which partition should contain the item. In that partition, several items could have the same partition key value. So DynamoDB stores the item among the others with the same partition key, in ascending order by sort key.

To read an item from the table, you must specify its partition key value and sort key value. DynamoDB calculates the partition key's hash value, yielding the partition in which the item can be found.

You can read multiple items from the table in a single operation (`Query`) if the items you want have the same partition key value. DynamoDB returns all of the items with that partition key value. Optionally, you can apply a condition to the sort key so that it returns only the items within a certain range of values.

Suppose that the *Pets* table has a composite primary key consisting of *AnimalType* (partition key) and *Name* (sort key). The following diagram shows DynamoDB writing an item with a partition key value of *Dog* and a sort key value of *Fido*.



To read that same item from the *Pets* table, DynamoDB calculates the hash value of *Dog*, yielding the partition in which these items are stored. DynamoDB then scans the sort key attribute values until it finds *Fido*.

To read all of the items with an *AnimalType* of *Dog*, you can issue a `Query` operation without specifying a sort key condition. By default, the items are returned in the order that they are stored (that is, in ascending order by sort key). Optionally, you can request descending order instead.

To query only some of the *Dog* items, you can apply a condition to the sort key (for example, only the *Dog* items where *Name* begins with a letter that is within the range `A` through `K`).

> **Note**
>
> In a DynamoDB table, there is no upper limit on the number of distinct sort key values per partition key value. If you needed to store many billions of *Dog* items in the *Pets* table, DynamoDB would allocate enough storage to handle this requirement automatically.

Learn how to go from SQL to NoSQL

If you are an application developer, you might have some experience using a relational database management system (RDBMS) and Structured Query Language (SQL). As you begin working with Amazon DynamoDB, you will encounter many similarities, but also many things that are different. *NoSQL* is a term used to describe nonrelational database systems that are highly available, scalable, and optimized for high performance. Instead of the relational model, NoSQL databases (like DynamoDB) use alternate models for data management, such as key-value pairs or document storage. For more information, see What is NoSQL?.

Amazon DynamoDB supports PartiQL, an open-source, SQL-compatible query language that makes it easy for you to efficiently query data, regardless of where or in what format it is stored. With PartiQL, you can easily process structured data from relational databases, semi-structured and nested data in open data formats, and even schema-less data in NoSQL or document databases that allow different attributes for different rows. For more information, see PartiQL query language.

The following sections describe common database tasks, comparing and contrasting SQL statements with their equivalent DynamoDB operations.

> **Note**
>
> The SQL examples in this section are compatible with the MySQL RDBMS.
> The DynamoDB examples in this section show the name of the DynamoDB operation, along with the parameters for that operation in JSON format.

Topics

- Choosing between relational (SQL) and NoSQL
- Differences in accessing a relational (SQL) database and DynamoDB

- Differences between a relational (SQL) database and DynamoDB when creating a table
- Differences between getting table information from a relational (SQL) database and DynamoDB
- Differences between a relational (SQL) database and DynamoDB when writing data to a table
- Differences between a relational (SQL) database and DynamoDB when reading data from a table
- Differences between a relational (SQL) database and DynamoDB when managing indexes
- Differences between a relational (SQL) database and DynamoDB when modifying data in a table
- Differences between a relational (SQL) database and DynamoDB when deleting data from a table
- Differences between a relational (SQL) database and DynamoDB when removing a table

Choosing between relational (SQL) and NoSQL

Today's applications have more demanding requirements than ever before. For example, an online game might start out with just a few users and a very small amount of data. However, if the game becomes successful, it can easily outstrip the resources of the underlying database management system. It is common for web-based applications to have hundreds, thousands, or millions of concurrent users, with terabytes or more of new data generated per day. Databases for such applications must handle tens (or hundreds) of thousands of reads and writes per second.

Amazon DynamoDB is well-suited for these kinds of workloads. As a developer, you can start small and gradually increase your utilization as your application becomes more popular. DynamoDB scales seamlessly to handle very large amounts of data and very large numbers of users.

For more information on traditional relational database modeling and how to adapt it for DynamoDB, see Best practices for modeling relational data in DynamoDB.

The following table shows some high-level differences between a relational database management system (RDBMS) and DynamoDB.

Characteristic	Relational database management system (RDBMS)	Amazon DynamoDB
Optimal Workloads	Ad hoc queries; data warehousing; OLAP (online analytical processing).	Web-scale applications, including social networks, gaming, media sharing, and Internet of Things (IoT).

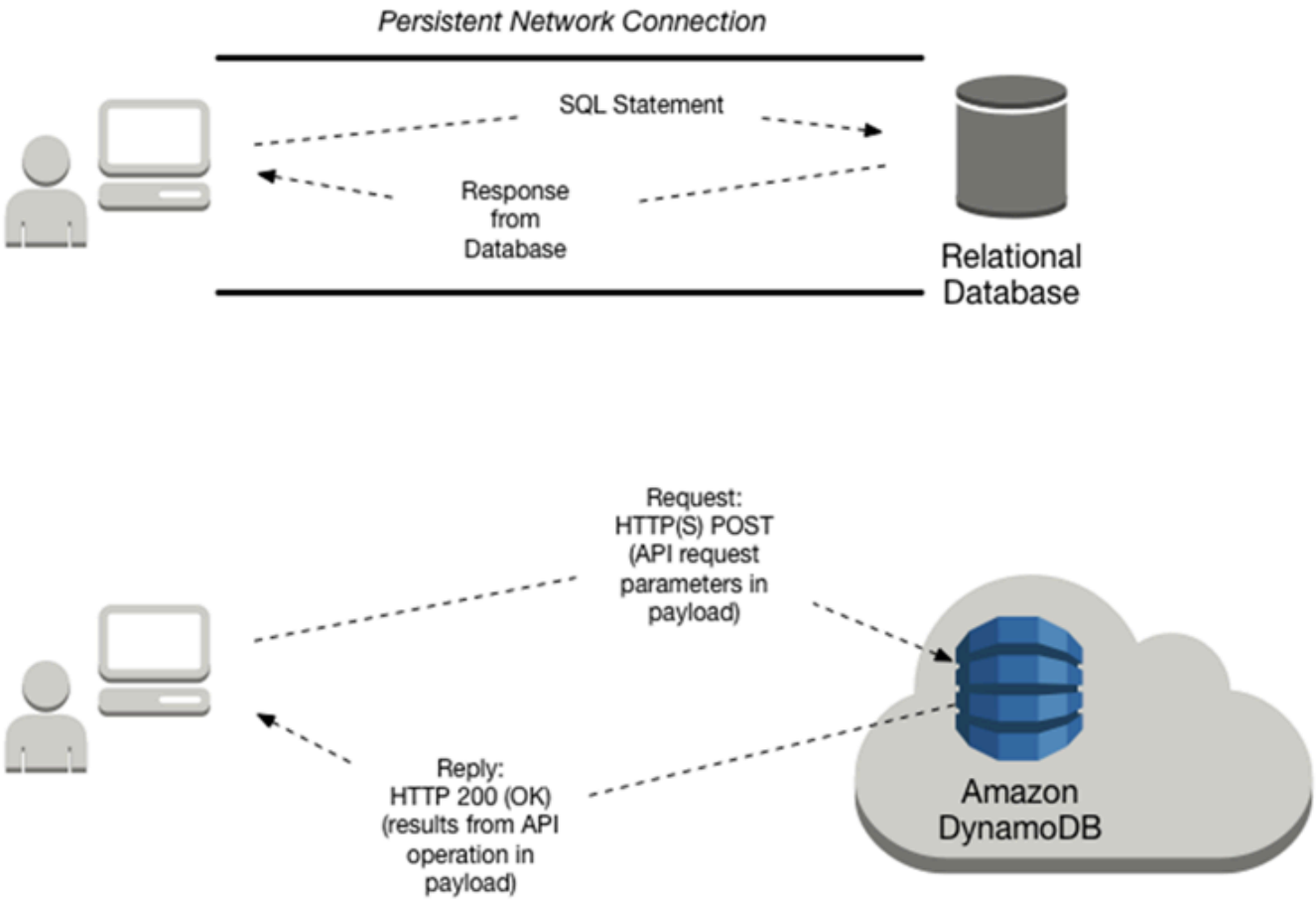
Characteristic	Relational database management system (RDBMS)	Amazon DynamoDB
Data Model	The relational model requires a well-defined schema, where data is normalized into tables, rows, and columns. In addition, all of the relationships are defined among tables, columns, indexes, and other database elements.	DynamoDB is schemaless. Every table must have a primary key to uniquely identify each data item, but there are no similar constraints on other non-key attributes. DynamoDB can manage structured or semistructured data, including JSON documents.
Data Access	SQL is the standard for storing and retrieving data. Relational databases offer a rich set of tools for simplifying the development of database-driven applications, but all of these tools use SQL.	You can use the AWS Management Console, the AWS CLI, or NoSQL WorkBench to work with DynamoDB and perform ad hoc tasks. PartiQL, a SQL-compatible query language, lets you select, insert, update, and delete data in DynamoDB. Applications can use the AWS software development kits (SDKs) to work with DynamoDB using object-based, document-centric, or low-level interfaces.

Characteristic	Relational database management system (RDBMS)	Amazon DynamoDB
Performance	Relational databases are optimized for storage, so performance generally depends on the disk subsystem. Developers and database administrators must optimize queries, indexes, and table structures in order to achieve peak performance.	DynamoDB is optimized for compute, so performan ce is mainly a function of the underlying hardware and network latency. As a managed service, DynamoDB insulates you and your applications from these implementation details, so that you can focus on designing and building robust, high-performance applications.
Scaling	It is easiest to scale up with faster hardware. It is also possible for database tables to span across multiple hosts in a distributed system, but this requires additiona l investment. Relational databases have maximum sizes for the number and size of files, which imposes upper limits on scalability.	DynamoDB is designed to scale out using distributed clusters of hardware. This design allows increased throughput without increased latency. Customers specify their throughput requireme nts, and DynamoDB allocates sufficient resources to meet those requirements. There are no upper limits on the number of items per table, nor the total size of that table.

Differences in accessing a relational (SQL) database and DynamoDB

Before your application can access a database, it must be *authenticated* to ensure that the application is allowed to use the database. It must be *authorized* so that the application can perform only the actions for which it has permissions.

The following diagram shows a client's interaction with a relational database and with Amazon DynamoDB.



The following table has more details about client interaction tasks.

Characteristic	Relational database management system (RDBMS)	Amazon DynamoDB
Tools for Accessing the Database	Most relational databases provide a command line	In most cases, you write application code. You can also

Characteristic	Relational database management system (RDBMS)	Amazon DynamoDB
	interface (CLI) so that you can enter ad hoc SQL statements and see the results immediately.	use the AWS Management Console, the AWS Command Line Interface (AWS CLI), or NoSQL Workbench to send ad hoc requests to DynamoDB and view the results. PartiQL, a SQL-compatible query language, lets you select, insert, update, and delete data in DynamoDB.
Connecting to the Database	An application program establishes and maintains a network connection with the database. When the application is finished, it terminates the connection.	DynamoDB is a web service, and interactions with it are stateless. Applications do not need to maintain persistent network connections. Instead, interaction with DynamoDB occurs using HTTP(S) requests and responses.
Authentication	An application cannot connect to the database until it is authenticated. The RDBMS can perform the authentication itself, or it can offload this task to the host operating system or a directory service.	Every request to DynamoDB must be accompanied by a cryptographic signature, which authenticates that particular request. The AWS SDKs provide all of the logic necessary for creating signatures and signing requests. For more information, see Signing AWS API requests in the *AWS General Reference*.

Characteristic	Relational database management system (RDBMS)	Amazon DynamoDB
Authorization	Applications can perform only the actions for which they have been authorized. Database administrators or application owners can use the SQL GRANT and REVOKE statements to control access to database objects (such as tables), data (such as rows within a table), or the ability to issue certain SQL statements.	In DynamoDB, authorization is handled by AWS Identity and Access Management (IAM). You can write an IAM policy to grant permissions on a DynamoDB resource (such as a table), and then allow users and roles to use that policy. IAM also features fine-grained access control for individual data items in DynamoDB tables. For more information, see Identity and Access Management for Amazon DynamoDB.
Sending a Request	The application issues a SQL statement for every database operation that it wants to perform. Upon receipt of the SQL statement, the RDBMS checks its syntax, creates a plan for performing the operation, and then runs the plan.	The application sends HTTP(S) requests to DynamoDB. The requests contain the name of the DynamoDB operation to perform, along with parameters. DynamoDB runs the request immediately.
Receiving a Response	The RDBMS returns the results from the SQL statement. If there is an error, the RDBMS returns an error status and message.	DynamoDB returns an HTTP(S) response containing the results of the operation. If there is an error, DynamoDB returns an HTTP error status and messages.

Differences between a relational (SQL) database and DynamoDB when creating a table

Tables are the fundamental data structures in relational databases and in Amazon DynamoDB. A relational database management system (RDBMS) requires you to define the table's schema when you create it. In contrast, DynamoDB tables are schemaless—other than the primary key, you do not need to define any extra attributes or data types when you create a table.

The following section compares how you would create a table with SQL to how you would create it with DynamoDB.

Topics

- Creating a table with SQL
- Creating a table with DynamoDB

Creating a table with SQL

With SQL you would use the `CREATE TABLE` statement to create a table, as shown in the following example.

```
CREATE TABLE Music (
    Artist VARCHAR(20) NOT NULL,
    SongTitle VARCHAR(30) NOT NULL,
    AlbumTitle VARCHAR(25),
    Year INT,
    Price FLOAT,
    Genre VARCHAR(10),
    Tags TEXT,
    PRIMARY KEY(Artist, SongTitle)
);
```

The primary key for this table consists of *Artist* and *SongTitle*.

You must define all of the table's columns and data types, and the table's primary key. (You can use the `ALTER TABLE` statement to change these definitions later, if necessary.)

Many SQL implementations let you define storage specifications for your table, as part of the `CREATE TABLE` statement. Unless you indicate otherwise, the table is created with default storage

settings. In a production environment, a database administrator can help determine the optimal storage parameters.

Creating a table with DynamoDB

Use the `CreateTable` operation to create a provisioned mode table, specifying parameters as shown following:

```
{
    TableName : "Music",
    KeySchema: [
        {
            AttributeName: "Artist",
            KeyType: "HASH" //Partition key
        },
        {
            AttributeName: "SongTitle",
            KeyType: "RANGE" //Sort key
        }
    ],
    AttributeDefinitions: [
        {
            AttributeName: "Artist",
            AttributeType: "S"
        },
        {
            AttributeName: "SongTitle",
            AttributeType: "S"
        }
    ],
    ProvisionedThroughput: {       // Only specified if using provisioned mode
        ReadCapacityUnits: 1,
        WriteCapacityUnits: 1
    }
}
```

The primary key for this table consists of *Artist* (partition key) and *SongTitle* (sort key).

You must provide the following parameters to `CreateTable`:

- `TableName` – Name of the table.

- `KeySchema` – Attributes that are used for the primary key. For more information, see Tables, items, and attributes and Primary key.
- `AttributeDefinitions` – Data types for the key schema attributes.
- `ProvisionedThroughput (for provisioned tables)` – Number of reads and writes per second that you need for this table. DynamoDB reserves sufficient storage and system resources so that your throughput requirements are always met. You can use the `UpdateTable` operation to change these later, if necessary. You do not need to specify a table's storage requirements because storage allocation is managed entirely by DynamoDB.

Differences between getting table information from a relational (SQL) database and DynamoDB

You can verify that a table has been created according to your specifications. In a relational database, all of the table's schema is shown. Amazon DynamoDB tables are schemaless, so only the primary key attributes are shown.

Topics

- Getting information about a table with SQL
- Getting information about a table in DynamoDB

Getting information about a table with SQL

Most relational database management systems (RDBMS) allow you to describe a table's structure—columns, data types, primary key definition, and so on. There is no standard way to do this in SQL. However, many database systems provide a `DESCRIBE` command. The following is an example from MySQL.

```
DESCRIBE Music;
```

This returns the structure of your table, with all of the column names, data types, and sizes.

```
+------------+-------------+------+-----+---------+-------+
| Field      | Type        | Null | Key | Default | Extra |
+------------+-------------+------+-----+---------+-------+
| Artist     | varchar(20) | NO   | PRI | NULL    |       |
| SongTitle  | varchar(30) | NO   | PRI | NULL    |       |
| AlbumTitle | varchar(25) | YES  |     | NULL    |       |
```

```
| Year       | int(11)     | YES  |     | NULL    |       |
| Price      | float       | YES  |     | NULL    |       |
| Genre      | varchar(10) | YES  |     | NULL    |       |
| Tags       | text        | YES  |     | NULL    |       |
+------------+-------------+------+-----+---------+-------+
```

The primary key for this table consists of *Artist* and *SongTitle*.

Getting information about a table in DynamoDB

DynamoDB has a `DescribeTable` operation, which is similar. The only parameter is the table name.

```
{
    TableName : "Music"
}
```

The reply from `DescribeTable` looks like the following.

```
{
  "Table": {
    "AttributeDefinitions": [
      {
        "AttributeName": "Artist",
        "AttributeType": "S"
      },
      {
        "AttributeName": "SongTitle",
        "AttributeType": "S"
      }
    ],
    "TableName": "Music",
    "KeySchema": [
      {
        "AttributeName": "Artist",
        "KeyType": "HASH"  //Partition key
      },
      {
        "AttributeName": "SongTitle",
        "KeyType": "RANGE"  //Sort key
      }
    ],
```

```
    ...
```

`DescribeTable` also returns information about indexes on the table, provisioned throughput settings, an approximate item count, and other metadata.

Differences between a relational (SQL) database and DynamoDB when writing data to a table

Relational database tables contain *rows* of data. Rows are composed of *columns*. Amazon DynamoDB tables contain *items*. Items are composed of *attributes*.

This section describes how to write one row (or item) to a table.

Topics

- Writing data to a table with SQL
- Writing data to a table in DynamoDB

Writing data to a table with SQL

A table in a relational database is a two-dimensional data structure composed of rows and columns. Some database management systems also provide support for semistructured data, usually with native JSON or XML data types. However, the implementation details vary among vendors.

In SQL, you would use the `INSERT` statement to add a row to a table.

```
INSERT INTO Music
    (Artist, SongTitle, AlbumTitle,
    Year, Price, Genre,
    Tags)
VALUES(
    'No One You Know', 'Call Me Today', 'Somewhat Famous',
    2015, 2.14, 'Country',
    '{"Composers": ["Smith", "Jones", "Davis"],"LengthInSeconds": 214}'
);
```

The primary key for this table consists of *Artist* and *SongTitle*. You must specify values for these columns.

> **Note**
>
> This example uses the *Tags* column to store semistructured data about the songs in the *Music* table. The *Tags* column is defined as type TEXT, which can store up to 65,535 characters in MySQL.

Writing data to a table in DynamoDB

In Amazon DynamoDB, you can use either the DynamoDB API or PartiQL (a SQL-compatible query language) to add an item to a table.

DynamoDB API

With the DynamoDB API, you use the `PutItem` operation to add an item to a table.

```
{
    TableName: "Music",
    Item: {
        "Artist":"No One You Know",
        "SongTitle":"Call Me Today",
        "AlbumTitle":"Somewhat Famous",
        "Year": 2015,
        "Price": 2.14,
        "Genre": "Country",
        "Tags": {
            "Composers": [
                  "Smith",
                  "Jones",
                  "Davis"
            ],
            "LengthInSeconds": 214
        }
    }
}
```

The primary key for this table consists of *Artist* and *SongTitle*. You must specify values for these attributes.

Here are some key things to know about this `PutItem` example:

- DynamoDB provides native support for documents, using JSON. This makes DynamoDB ideal for storing semistructured data, such as *Tags*. You can also retrieve and manipulate data from within JSON documents.
- The *Music* table does not have any predefined attributes, other than the primary key (*Artist* and *SongTitle*).
- Most SQL databases are transaction oriented. When you issue an `INSERT` statement, the data modifications are not permanent until you issue a `COMMIT` statement. With Amazon DynamoDB, the effects of a `PutItem` operation are permanent when DynamoDB replies with an HTTP 200 status code (`OK`).

The following are some other `PutItem` examples.

```
{
    TableName: "Music",
    Item: {
        "Artist": "No One You Know",
        "SongTitle": "My Dog Spot",
        "AlbumTitle":"Hey Now",
        "Price": 1.98,
        "Genre": "Country",
        "CriticRating": 8.4
    }
}
```

```
{
    TableName: "Music",
    Item: {
        "Artist": "No One You Know",
        "SongTitle": "Somewhere Down The Road",
        "AlbumTitle":"Somewhat Famous",
        "Genre": "Country",
        "CriticRating": 8.4,
        "Year": 1984
    }
}
```

```
{
    TableName: "Music",
    Item: {
```

```
        "Artist": "The Acme Band",
        "SongTitle": "Still In Love",
        "AlbumTitle":"The Buck Starts Here",
        "Price": 2.47,
        "Genre": "Rock",
        "PromotionInfo": {
            "RadioStationsPlaying":[
                 "KHCR", "KBQX", "WTNR", "WJJH"
            ],
            "TourDates": {
                "Seattle": "20150625",
                "Cleveland": "20150630"
            },
            "Rotation": "Heavy"
        }
    }
}
```

```
{
    TableName: "Music",
    Item: {
        "Artist": "The Acme Band",
        "SongTitle": "Look Out, World",
        "AlbumTitle":"The Buck Starts Here",
        "Price": 0.99,
        "Genre": "Rock"
    }
}
```

> **Note**
>
> In addition to `PutItem`, DynamoDB supports a `BatchWriteItem` operation for writing multiple items at the same time.

PartiQL for DynamoDB

With PartiQL, you use the `ExecuteStatement` operation to add an item to a table, using the PartiQL `Insert` statement.

```
INSERT into Music value {
```

```
    'Artist': 'No One You Know',
    'SongTitle': 'Call Me Today',
    'AlbumTitle': 'Somewhat Famous',
    'Year' : '2015',
    'Genre' : 'Acme'
}
```

The primary key for this table consists of *Artist* and *SongTitle*. You must specify values for these attributes.

> **Note**
>
> For code examples using `Insert` and `ExecuteStatement`, see PartiQL insert statements for DynamoDB.

Differences between a relational (SQL) database and DynamoDB when reading data from a table

With SQL, you use the `SELECT` statement to retrieve one or more rows from a table. You use the `WHERE` clause to determine the data that is returned to you.

This is different than using Amazon DynamoDB which provides the following operations for reading data:

- `ExecuteStatement` retrieves a single or multiple items from a table. `BatchExecuteStatement` retrieves multiple items from different tables in a single operation. Both of these operations use PartiQL, a SQL-compatible query language.
- `GetItem` – Retrieves a single item from a table. This is the most efficient way to read a single item because it provides direct access to the physical location of the item. (DynamoDB also provides the `BatchGetItem` operation, allowing you to perform up to 100 `GetItem` calls in a single operation.)
- `Query` – Retrieves all of the items that have a specific partition key. Within those items, you can apply a condition to the sort key and retrieve only a subset of the data. `Query` provides quick, efficient access to the partitions where the data is stored. (For more information, see Partitions and data distribution in DynamoDB.)
- `Scan` – Retrieves all of the items in the specified table. (This operation should not be used with large tables because it can consume large amounts of system resources.)

> **Note**
>
> With a relational database, you can use the SELECT statement to join data from multiple tables and return the results. Joins are fundamental to the relational model. To ensure that joins run efficiently, the database and its applications should be performance-tuned on an ongoing basis. DynamoDB is a non-relational NoSQL database that does not support table joins. Instead, applications read data from one table at a time.

The following sections describe different use cases for reading data, and how to perform these tasks with a relational database and with DynamoDB.

Topics

- Differences in reading an item using its primary key
- Differences in querying a table
- Differences in scanning a table

Differences in reading an item using its primary key

One common access pattern for databases is to read a single item from a table. You have to specify the primary key of the item you want.

Topics

- Reading an item using its primary key with SQL
- Reading an item using its primary key in DynamoDB

Reading an item using its primary key with SQL

In SQL, you would use the SELECT statement to retrieve data from a table. You can request one or more columns in the result (or all of them, if you use the * operator). The WHERE clause determines which rows to return.

The following is a SELECT statement to retrieve a single row from the *Music* table. The WHERE clause specifies the primary key values.

```
SELECT *
```

```
FROM Music
WHERE Artist='No One You Know' AND SongTitle = 'Call Me Today'
```

You can modify this query to retrieve only a subset of the columns.

```
SELECT AlbumTitle, Year, Price
FROM Music
WHERE Artist='No One You Know' AND SongTitle = 'Call Me Today'
```

Note that the primary key for this table consists of *Artist* and *SongTitle*.

Reading an item using its primary key in DynamoDB

In Amazon DynamoDB, you can use either the DynamoDB API or PartiQL (a SQL-compatible query language) to read an item from a table.

DynamoDB API

With the DynamoDB API, you use the `PutItem` operation to add an item to a table.

DynamoDB provides the `GetItem` operation for retrieving an item by its primary key. `GetItem` is highly efficient because it provides direct access to the physical location of the item. (For more information, see Partitions and data distribution in DynamoDB.)

By default, `GetItem` returns the entire item with all of its attributes.

```
{
    TableName: "Music",
    Key: {
        "Artist": "No One You Know",
        "SongTitle": "Call Me Today"
    }
}
```

You can add a `ProjectionExpression` parameter to return only some of the attributes.

```
{
    TableName: "Music",
    Key: {
        "Artist": "No One You Know",
```

```
        "SongTitle": "Call Me Today"
    },
    "ProjectionExpression": "AlbumTitle, Year, Price"
}
```

Note that the primary key for this table consists of *Artist* and *SongTitle*.

The DynamoDB `GetItem` operation is very efficient. It uses the primary key values to determine the exact storage location of the item in question, and retrieves it directly from there. The SQL `SELECT` statement is similarly efficient, in the case of retrieving items by primary key values.

The SQL `SELECT` statement supports many kinds of queries and table scans. DynamoDB provides similar functionality with its `Query` and `Scan` operations, which are described in Differences in querying a table and Differences in scanning a table.

The SQL `SELECT` statement can perform table joins, allowing you to retrieve data from multiple tables at the same time. Joins are most effective where the database tables are normalized and the relationships among the tables are clear. However, if you join too many tables in one `SELECT` statement application performance can be affected. You can work around such issues by using database replication, materialized views, or query rewrites.

DynamoDB is a nonrelational database and doesn't support table joins. If you are migrating an existing application from a relational database to DynamoDB, you need to denormalize your data model to eliminate the need for joins.

PartiQL for DynamoDB

With PartiQL, you use the `ExecuteStatement` operation to read an item from a table, using the PartiQL `Select` statement.

```
SELECT AlbumTitle, Year, Price
FROM Music
WHERE Artist='No One You Know' AND SongTitle = 'Call Me Today'
```

Note that the primary key for this table consists of Artist and SongTitle.

> **Note**
>
> The select PartiQL statement can also be used to Query or Scan a DynamoDB table

For code examples using `Select` and `ExecuteStatement`, see PartiQL select statements for DynamoDB.

Differences in querying a table

Another common access pattern is reading multiple items from a table, based on your query criteria.

Topics

- Querying a table with SQL
- Querying a table in DynamoDB

Querying a table with SQL

When using SQL the `SELECT` statement lets you query on key columns, non-key columns, or any combination. The `WHERE` clause determines which rows are returned, as shown in the following examples.

```
/* Return a single song, by primary key */

SELECT * FROM Music
WHERE Artist='No One You Know' AND SongTitle = 'Call Me Today';
```

```
/* Return all of the songs by an artist */

SELECT * FROM Music
WHERE Artist='No One You Know';
```

```
/* Return all of the songs by an artist, matching first part of title */

SELECT * FROM Music
WHERE Artist='No One You Know' AND SongTitle LIKE 'Call%';
```

```
/* Return all of the songs by an artist, only if the price is less than 1.00 */

SELECT * FROM Music
WHERE Artist='No One You Know'
```

```
AND Price < 1.00;
```

Note that the primary key for this table consists of *Artist* and *SongTitle*.

Querying a table in DynamoDB

In Amazon DynamoDB, you can use either the DynamoDB API or PartiQL (a SQL-compatible query language) to query an item from a table.

DynamoDB API

With Amazon DynamoDB, you can use the `Query` operation to retrieve data in a similar fashion. The `Query` operation provides quick, efficient access to the physical locations where the data is stored. For more information, see Partitions and data distribution in DynamoDB.

You can use `Query` with any table or secondary index. You must specify an equality condition for the partition key's value, and you can optionally provide another condition for the sort key attribute if it is defined.

The `KeyConditionExpression` parameter specifies the key values that you want to query. You can use an optional `FilterExpression` to remove certain items from the results before they are returned to you.

In DynamoDB, you must use `ExpressionAttributeValues` as placeholders in expression parameters (such as `KeyConditionExpression` and `FilterExpression`). This is analogous to the use of *bind variables* in relational databases, where you substitute the actual values into the `SELECT` statement at runtime.

Note that the primary key for this table consists of *Artist* and *SongTitle*.

The following are some DynamoDB `Query` examples.

```
// Return a single song, by primary key

{
    TableName: "Music",
    KeyConditionExpression: "Artist = :a and SongTitle = :t",
    ExpressionAttributeValues: {
        ":a": "No One You Know",
        ":t": "Call Me Today"
```

```
    }
}
```

```
// Return all of the songs by an artist

{
    TableName: "Music",
    KeyConditionExpression: "Artist = :a",
    ExpressionAttributeValues: {
        ":a": "No One You Know"
    }
}
```

```
// Return all of the songs by an artist, matching first part of title

{
    TableName: "Music",
    KeyConditionExpression: "Artist = :a and begins_with(SongTitle, :t)",
    ExpressionAttributeValues: {
        ":a": "No One You Know",
        ":t": "Call"
    }
}
```

```
// Return all of the songs by an artist, only if the price is less than 1.00

{
    TableName: "Music",
    KeyConditionExpression: "Artist = :a",
    FilterExpression: "Price < :p",
    ExpressionAttributeValues: {
        ":a": "No One You Know",
        ":p": 1.00
    }
}
```

Note

A `FilterExpression` is applied after the `Query` reads matching items, so it does not reduce the read capacity consumed. Where possible, model your data so that range

conditions use `KeyConditionExpression` on the sort key for efficient queries. For more information, see Querying tables in DynamoDB.

PartiQL for DynamoDB

With PartiQL, you can perform a query by using the `ExecuteStatement` operation and the `Select` statement on the partition key.

```
SELECT AlbumTitle, Year, Price
FROM Music
WHERE Artist='No One You Know'
```

Using the `SELECT` statement in this way returns all the songs associated with this particular `Artist`.

For code examples using `Select` and `ExecuteStatement`, see PartiQL select statements for DynamoDB.

Differences in scanning a table

In SQL, a `SELECT` statement without a `WHERE` clause will return every row in a table. In Amazon DynamoDB, the `Scan` operation does the same thing. In both cases, you can retrieve all of the items or just some of them.

Whether you are using a SQL or a NoSQL database, scans should be used sparingly because they can consume large amounts of system resources. Sometimes a scan is appropriate (such as scanning a small table) or unavoidable (such as performing a bulk export of data). However, as a general rule, you should design your applications to avoid performing scans. For more information, see Querying tables in DynamoDB.

Note

Doing a bulk export also creates at least 1 file per partition. All of the items in each file are from that particular partition's hashed keyspace.

Topics

- Scanning a table with SQL

- Scanning a table in DynamoDB

Scanning a table with SQL

When using SQL you can scan a table and retrieve all of its data by using a `SELECT` statement without specifying a `WHERE` clause. You can request one or more columns in the result. Or you can request all of them if you use the wildcard character (*).

The following are examples of using a `SELECT` statement.

```
/* Return all of the data in the table */
SELECT * FROM Music;
```

```
/* Return all of the values for Artist and Title */
SELECT Artist, Title FROM Music;
```

Scanning a table in DynamoDB

In Amazon DynamoDB, you can use either the DynamoDB API or PartiQL (a SQL-compatible query language) to perform a scan on a table.

DynamoDB API

With the DynamoDB API, you use the `Scan` operation to return one or more items and item attributes by accessing every item in a table or a secondary index.

```
// Return all of the data in the table
{
    TableName:  "Music"
}
```

```
// Return all of the values for Artist and Title
{
    TableName:  "Music",
    ProjectionExpression: "Artist, Title"
}
```

The `Scan` operation also provides a `FilterExpression` parameter, which you can use to discard items that you do not want to appear in the results. A `FilterExpression` is

applied after the scan is performed, but before the results are returned to you. (This is not recommended with large tables. You are still charged for the entire `Scan`, even if only a few matching items are returned.)

PartiQL for DynamoDB

With PartiQL, you perform a scan by using the `ExecuteStatement` operation to return all the contents for a table using the `Select` statement.

```
SELECT AlbumTitle, Year, Price
FROM Music
```

Note that this statement will return all items for in the Music table.

For code examples using `Select` and `ExecuteStatement`, see PartiQL select statements for DynamoDB.

Differences between a relational (SQL) database and DynamoDB when managing indexes

Indexes give you access to alternate query patterns, and can speed up queries. This section compares and contrasts index creation and usage in SQL and Amazon DynamoDB.

Whether you are using a relational database or DynamoDB, you should be judicious with index creation. Whenever a write occurs on a table, all of the table's indexes must be updated. In a write-heavy environment with large tables, this can consume large amounts of system resources. In a read-only or read-mostly environment, this is not as much of a concern. However, you should ensure that the indexes are actually being used by your application, and not simply taking up space.

Topics

- Differences between a relational (SQL) database and DynamoDB when creating an index
- Differences between a relational (SQL) database and DynamoDB when querying and scanning an index

Differences between a relational (SQL) database and DynamoDB when creating an index

Compare the `CREATE INDEX` statement in SQL with the `UpdateTable` operation in Amazon DynamoDB.

Topics

- Creating an index with SQL
- Creating an index in DynamoDB

Creating an index with SQL

In a relational database, an index is a data structure that lets you perform fast queries on different columns in a table. You can use the `CREATE INDEX` SQL statement to add an index to an existing table, specifying the columns to be indexed. After the index has been created, you can query the data in the table as usual, but now the database can use the index to quickly find the specified rows in the table instead of scanning the entire table.

After you create an index, the database maintains it for you. Whenever you modify data in the table, the index is automatically modified to reflect changes in the table.

In MySQL, you would create an index like the following.

```
CREATE INDEX GenreAndPriceIndex
ON Music (genre, price);
```

Creating an index in DynamoDB

In DynamoDB, you can create and use a *secondary index* for similar purposes.

Indexes in DynamoDB are different from their relational counterparts. When you create a secondary index, you must specify its key attributes—a partition key and a sort key. After you create the secondary index, you can `Query` it or `Scan` it just as you would with a table. DynamoDB does not have a query optimizer, so a secondary index is only used when you `Query` it or `Scan` it.

DynamoDB supports two different kinds of indexes:

- Global secondary indexes – The primary key of the index can be any two attributes from its table.

- Local secondary indexes – The partition key of the index must be the same as the partition key of its table. However, the sort key can be any other attribute.

DynamoDB ensures that the data in a secondary index is eventually consistent with its table. You can request strongly consistent `Query` or `Scan` operations on a table or a local secondary index. However, global secondary indexes support only eventual consistency.

You can add a global secondary index to an existing table, using the `UpdateTable` operation and specifying `GlobalSecondaryIndexUpdates`.

```
{
    TableName: "Music",
    AttributeDefinitions:[
        {AttributeName: "Genre", AttributeType: "S"},
        {AttributeName: "Price", AttributeType: "N"}
    ],
    GlobalSecondaryIndexUpdates: [
        {
            Create: {
                IndexName: "GenreAndPriceIndex",
                KeySchema: [
                    {AttributeName: "Genre", KeyType: "HASH"}, //Partition key
                    {AttributeName: "Price", KeyType: "RANGE"}, //Sort key
                ],
                Projection: {
                    "ProjectionType": "ALL"
                },
                ProvisionedThroughput: {                                // Only
 specified if using provisioned mode
                    "ReadCapacityUnits": 1,"WriteCapacityUnits": 1
                }
            }
        }
    ]
}
```

You must provide the following parameters to `UpdateTable`:

- `TableName` – The table that the index will be associated with.
- `AttributeDefinitions` – The data types for the key schema attributes of the index.
- `GlobalSecondaryIndexUpdates` – Details about the index you want to create:

- `IndexName` – A name for the index.
- `KeySchema` – The attributes that are used for the index's primary key.
- `Projection` – Attributes from the table that are copied to the index. In this case, `ALL` means that all of the attributes are copied.
- `ProvisionedThroughput (for provisioned tables)` – The number of reads and writes per second that you need for this index. (This is separate from the provisioned throughput settings of the table.)

Part of this operation involves backfilling data from the table into the new index. During backfilling, the table remains available. However, the index is not ready until its `Backfilling` attribute changes from true to false. You can use the `DescribeTable` operation to view this attribute.

Differences between a relational (SQL) database and DynamoDB when querying and scanning an index

Compare querying and scanning an index using the SELECT statement in SQL with the `Query` and `Scan` operations in Amazon DynamoDB.

Topics

- Querying and scanning an index with SQL
- Querying and scanning an index in DynamoDB

Querying and scanning an index with SQL

In a relational database, you do not work directly with indexes. Instead, you query tables by issuing `SELECT` statements, and the query optimizer can make use of any indexes.

A *query optimizer* is a relational database management system (RDBMS) component that evaluates the available indexes and determines whether they can be used to speed up a query. If the indexes can be used to speed up a query, the RDBMS accesses the index first and then uses it to locate the data in the table.

Here are some SQL statements that can use *GenreAndPriceIndex* to improve performance. We assume that the *Music* table has enough data in it that the query optimizer decides to use this index, rather than simply scanning the entire table.

```
/* All of the rock songs */

SELECT * FROM Music
WHERE Genre = 'Rock';
```

```
/* All of the cheap country songs */

SELECT Artist, SongTitle, Price FROM Music
WHERE Genre = 'Country' AND Price < 0.50;
```

Querying and scanning an index in DynamoDB

In DynamoDB, you perform `Query` and `Scan` operations directly on the index, in the same way that you would on a table. You can use either the DynamoDB API or PartiQL (a SQL-compatible query language) to query or scan the index. You must specify both `TableName` and `IndexName`.

The following are some queries on *GenreAndPriceIndex* in DynamoDB. (The key schema for this index consists of *Genre* and *Price*.)

DynamoDB API

```
// All of the rock songs

{
    TableName: "Music",
    IndexName: "GenreAndPriceIndex",
    KeyConditionExpression: "Genre = :genre",
    ExpressionAttributeValues: {
        ":genre": "Rock"
    },
};
```

This example uses a `ProjectionExpression` to indicate that you only want some of the attributes, rather than all of them, to appear in the results.

```
// All of the cheap country songs

{
    TableName: "Music",
    IndexName: "GenreAndPriceIndex",
```

```
    KeyConditionExpression: "Genre = :genre and Price < :price",
    ExpressionAttributeValues: {
        ":genre": "Country",
        ":price": 0.50
    },
    ProjectionExpression: "Artist, SongTitle, Price"
};
```

The following is a scan on *GenreAndPriceIndex*.

```
// Return all of the data in the index

{
    TableName:  "Music",
    IndexName: "GenreAndPriceIndex"
}
```

PartiQL for DynamoDB

With PartiQL, you use the PartiQL `Select` statement to perform queries and scans on the index.

```
// All of the rock songs

SELECT *
FROM Music.GenreAndPriceIndex
WHERE Genre = 'Rock'
```

```
// All of the cheap country songs

SELECT *
FROM Music.GenreAndPriceIndex
WHERE Genre = 'Rock' AND Price < 0.50
```

The following is a scan on *GenreAndPriceIndex*.

```
// Return all of the data in the index

SELECT *
FROM Music.GenreAndPriceIndex
```

> **Note**
>
> For code examples using `Select`, see PartiQL select statements for DynamoDB.

Differences between a relational (SQL) database and DynamoDB when modifying data in a table

The SQL language provides the `UPDATE` statement for modifying data. Amazon DynamoDB uses the `UpdateItem` operation to accomplish similar tasks.

Topics

- Modifying data in a table with SQL
- Modifying data in a table in DynamoDB

Modifying data in a table with SQL

In SQL, you would use the `UPDATE` statement to modify one or more rows. The `SET` clause specifies new values for one or more columns, and the `WHERE` clause determines which rows are modified. The following is an example.

```
UPDATE Music
SET RecordLabel = 'Global Records'
WHERE Artist = 'No One You Know' AND SongTitle = 'Call Me Today';
```

If no rows match the `WHERE` clause, the `UPDATE` statement has no effect.

Modifying data in a table in DynamoDB

In DynamoDB, you can use either the DynamoDB API or PartiQL (a SQL-compatible query language) to modify a single item. If you want to modify multiple items, you must use multiple operations.

DynamoDB API

With the DynamoDB API, you use the `UpdateItem` operation to modify a single item.

```
{
```

```
    TableName: "Music",
    Key: {
        "Artist":"No One You Know",
        "SongTitle":"Call Me Today"
    },
    UpdateExpression: "SET RecordLabel = :label",
    ExpressionAttributeValues: {
        ":label": "Global Records"
    }
}
```

You must specify the `Key` attributes of the item to be modified and an `UpdateExpression` to specify attribute values. `UpdateItem` behaves like an "upsert" operation. The item is updated if it exists in the table, but if not, a new item is added (inserted).

`UpdateItem` supports *conditional writes*, where the operation succeeds only if a specific `ConditionExpression` evaluates to true. For example, the following `UpdateItem` operation does not perform the update unless the price of the song is greater than or equal to 2.00.

```
{
    TableName: "Music",
    Key: {
        "Artist":"No One You Know",
        "SongTitle":"Call Me Today"
    },
    UpdateExpression: "SET RecordLabel = :label",
    ConditionExpression: "Price >= :p",
    ExpressionAttributeValues: {
        ":label": "Global Records",
        ":p": 2.00
    }
}
```

`UpdateItem` also supports *atomic counters*, or attributes of type `Number` that can be incremented or decremented. Atomic counters are similar in many ways to sequence generators, identity columns, or autoincrement fields in SQL databases.

The following is an example of an `UpdateItem` operation to initialize a new attribute (*Plays*) to keep track of the number of times a song has been played.

```
{
```

```
    TableName: "Music",
    Key: {
        "Artist":"No One You Know",
        "SongTitle":"Call Me Today"
    },
    UpdateExpression: "SET Plays = :val",
    ExpressionAttributeValues: {
        ":val": 0
    },
    ReturnValues: "UPDATED_NEW"
}
```

The `ReturnValues` parameter is set to `UPDATED_NEW`, which returns the new values of any attributes that were updated. In this case, it returns 0 (zero).

Whenever someone plays this song, we can use the following `UpdateItem` operation to increment *Plays* by one.

```
{
    TableName: "Music",
    Key: {
        "Artist":"No One You Know",
        "SongTitle":"Call Me Today"
    },
    UpdateExpression: "SET Plays = Plays + :incr",
    ExpressionAttributeValues: {
        ":incr": 1
    },
    ReturnValues: "UPDATED_NEW"
}
```

PartiQL for DynamoDB

With PartiQL, you use the `ExecuteStatement` operation to modify an item in a table, using the PartiQL `Update` statement.

The primary key for this table consists of *Artist* and *SongTitle*. You must specify values for these attributes.

```
UPDATE Music
SET RecordLabel ='Global Records'
WHERE Artist='No One You Know' AND SongTitle='Call Me Today'
```

You can also modify multiple fields at once, such as in the following example.

```
UPDATE Music
SET RecordLabel = 'Global Records'
SET AwardsWon = 10
WHERE Artist ='No One You Know' AND SongTitle='Call Me Today'
```

`Update` also supports *atomic counters*, or attributes of type `Number` that can be incremented or decremented. Atomic counters are similar in many ways to sequence generators, identity columns, or autoincrement fields in SQL databases.

The following is an example of an `Update` statement to initialize a new attribute (*Plays*) to keep track of the number of times a song has been played.

```
UPDATE Music
SET Plays = 0
WHERE Artist='No One You Know' AND SongTitle='Call Me Today'
```

Whenever someone plays this song, we can use the following `Update` statement to increment *Plays* by one.

```
UPDATE Music
SET Plays = Plays + 1
WHERE Artist='No One You Know' AND SongTitle='Call Me Today'
```

> **Note**
>
> For code examples using `Update` and `ExecuteStatement`, see PartiQL update statements for DynamoDB.

Differences between a relational (SQL) database and DynamoDB when deleting data from a table

In SQL, the `DELETE` statement removes one or more rows from a table. Amazon DynamoDB uses the `DeleteItem` operation to delete one item at a time.

Topics

- Deleting data from a table with SQL

- Deleting data from a table in DynamoDB

Deleting data from a table with SQL

In SQL, you use the `DELETE` statement to delete one or more rows. The `WHERE` clause determines the rows that you want to modify. The following is an example.

```
DELETE FROM Music
WHERE Artist = 'The Acme Band' AND SongTitle = 'Look Out, World';
```

You can modify the `WHERE` clause to delete multiple rows. For example, you could delete all of the songs by a particular artist, as shown in the following example.

```
DELETE FROM Music WHERE Artist = 'The Acme Band'
```

Deleting data from a table in DynamoDB

In DynamoDB, you can use either the DynamoDB API or PartiQL (a SQL-compatible query language) to delete a single item. If you want to modify multiple items, you must use multiple operations.

DynamoDB API

With the DynamoDB API, you use the `DeleteItem` operation to delete data from a table, one item at a time. You must specify the item's primary key values.

```
{
    TableName: "Music",
    Key: {
        Artist: "The Acme Band",
        SongTitle: "Look Out, World"
    }
}
```

> **Note**
>
> In addition to `DeleteItem`, Amazon DynamoDB supports a `BatchWriteItem` operation for deleting multiple items at the same time.

`DeleteItem` supports *conditional writes*, where the operation succeeds only if a specific `ConditionExpression` evaluates to true. For example, the following `DeleteItem` operation deletes the item only if it has a *RecordLabel* attribute.

```
{
    TableName: "Music",
    Key: {
        Artist: "The Acme Band",
        SongTitle: "Look Out, World"
    },
   ConditionExpression: "attribute_exists(RecordLabel)"
}
```

PartiQL for DynamoDB

With PartiQL, you use the `Delete` statement through the `ExecuteStatement` operation to delete data from a table, one item at a time. You must specify the item's primary key values.

The primary key for this table consists of *Artist* and *SongTitle*. You must specify values for these attributes.

```
DELETE FROM Music
WHERE Artist = 'Acme Band' AND SongTitle = 'PartiQL Rocks'
```

You can also specify additional conditions for the operation. The following `DELETE` operation only deletes the item if it has more than 11 *Awards*.

```
DELETE FROM Music
WHERE Artist = 'Acme Band' AND SongTitle = 'PartiQL Rocks' AND Awards > 11
```

> **Note**
>
> For code examples using `DELETE` and `ExecuteStatement`, see PartiQL delete statements for DynamoDB.

Differences between a relational (SQL) database and DynamoDB when removing a table

In SQL, you use the `DROP TABLE` statement to remove a table. In Amazon DynamoDB, you use the `DeleteTable` operation.

Topics

- Removing a table with SQL
- Removing a table in DynamoDB

Removing a table with SQL

When you no longer need a table and want to discard it permanently, you would use the `DROP TABLE` statement in SQL.

```
DROP TABLE Music;
```

After a table is dropped, it cannot be recovered. (Some relational databases do allow you to undo a `DROP TABLE` operation, but this is vendor-specific functionality and it is not widely implemented.)

Removing a table in DynamoDB

In DynamoDB, `DeleteTable` is a similar operation. In the following example, the table is permanently deleted.

```
{
    TableName: "Music"
}
```

Amazon DynamoDB learning resources and tools

You can use the following additional resources to understand and work with DynamoDB.

Topics

- Tools for coding and visualization
- Prescriptive Guidance articles
- Knowledge Center articles

- Blog posts, repositories, and guides
- Data modeling and design pattern presentations
- Training courses

Tools for coding and visualization

You can use the following coding and visualization tools to work with DynamoDB:

- NoSQL Workbench for Amazon DynamoDB – A unified, visual tool that helps you design, create, query, and manage DynamoDB tables. It provides data modeling, data visualization, and query development features.
- Dynobase – A desktop tool that makes it easy to see your DynamoDB tables and work with them, create app code, and edit records with real-time validation.
- DynamoDB Toolbox – A project from Jeremy Daly that provides helpful utilities for working with data modeling and JavaScript and Node.js.
- DynamoDB Streams Processor – A simple tool that you can use to work with DynamoDB streams.

Prescriptive Guidance articles

AWS Prescriptive Guidance provides time-tested strategies, guides, and patterns to help accelerate your projects. These resources were developed by AWS technology experts and the global community of AWS Partners, based on their years of experience helping customers achieve their business objectives.

Data modeling and migration

- A hierarchical data model in DynamoDB
- Modeling data with DynamoDB
- Migrate an Oracle database to DynamoDB using AWS DMS

Global tables

- Using Amazon DynamoDB global tables

Serverless

- Implement the serverless saga pattern with AWS Step Functions

SaaS architecture

- Manage tenants across multiple SaaS products on a single control plane
- Tenant onboarding in SaaS architecture for the silo model using C# and AWS CDK

Data protection and data movement

- Configure cross-account access to Amazon DynamoDB
- Full table copy options for DynamoDB
- Disaster recovery strategy for databases on AWS

Miscellaneous

- Help enforce tagging in DynamoDB

Prescriptive guidance video walkthroughs

- Using Serverless Architecture to Create Data Pipelines
- Novartis - Buying Engine: AI-powered Procurement Portal
- Veritiv: Enable Insights to Forecast Sales Demand on AWS Data Lakes
- mimik: Hybrid Edge Cloud Leveraging AWS to Support Edge Microservice Mesh
- Change Data Capture with Amazon DynamoDB

For additional Prescriptive Guidance articles and videos for DynamoDB, see Prescriptive Guidance.

Knowledge Center articles

The AWS Knowledge Center articles and videos cover the most frequent questions and requests that we receive from AWS customers. The following are some current Knowledge Center articles on specific tasks that relate to DynamoDB:

Cost optimization

- How do I optimize costs with Amazon DynamoDB?

Throttling and latency

- How can I troubleshoot high latency on an Amazon DynamoDB table?
- Why is my DynamoDB table being throttled?
- Why is my on-demand DynamoDB table being throttled?

Pagination

- How do I implement pagination in DynamoDB

Transactions

- Why is my `TransactWriteItems` API call failing in DynamoDB

Troubleshooting

- How do I resolve issues with DynamoDB auto scaling?
- How do I troubleshoot HTTP 4XX errors in DynamoDB

For additional articles and videos for DynamoDB, see the Knowledge Center articles.

Blog posts, repositories, and guides

In addition to the DynamoDB Developer Guide, there are many useful resources for working with DynamoDB. Here are some selected blog posts, repositories, and guides for working with DynamoDB:

- AWS repository of DynamoDB code examples in various AWS SDK languages: Node.js, Java, Python, .Net, Go, and Rust.
- The DynamoDB Book – A comprehensive guide from Alex DeBrie that teaches a strategy-driven approach to data modeling with DynamoDB.
- DynamoDB guide – An open guide from Alex DeBrie that walks through the basic concepts and advanced features of the DynamoDB NoSQL database.
- How to switch from RDBMS to DynamoDB in 20 easy steps – A list of useful steps for learning data modeling from Jeremy Daly.

- DynamoDB JavaScript DocumentClient cheat sheet – A cheat sheet to help you get started building applications with DynamoDB in a Node.js or JavaScript environment.
- DynamoDB Core Concept Videos – This playlist covers many of the core concepts of DynamoDB.

Data modeling and design pattern presentations

You can use the following resources on data modeling and design patterns to help you get the most out of DynamoDB:

- AWS re:Invent 2019: Data modeling with DynamoDB
  - A talk by Alex DeBrie that helps you started with the principles of DynamoDB data modeling.
- AWS re:Invent 2020: Data modeling with DynamoDB – Part 1
- AWS re:Invent 2020: Data modeling with DynamoDB – Part 2
- AWS re:Invent 2017: Advanced design patterns
- AWS re:Invent 2018: Advanced design patterns
- AWS re:Invent 2019: Advanced design patterns
  - Jeremy Daly shares his 12 key takeaways from this session.
- AWS re:Invent 2020: DynamoDB advanced design patterns – Part 1
- AWS re:Invent 2020: DynamoDB advanced design patterns – Part 2
- DynamoDB Office Hours on Twitch

> **Note**
>
> Each session covers different use cases and examples.

Training courses

There are many different training courses and educational options for learning more about DynamoDB. Here are some current examples:

- Developing with Amazon DynamoDB – Designed by AWS to take you from beginner to expert in developing real-world applications with data modeling for Amazon DynamoDB.
- DynamoDB deep dive course – A course from Pluralsight.

- Amazon DynamoDB: Building NoSQL database-driven applications – A course from the AWS Training and Certification team hosted on edX.

DynamoDB reads and writes

DynamoDB reads and writes refer to the operations that retrieve data from a table (reads) and insert, update, or delete data in a table (writes). When you work with DynamoDB, it's essential to understand the concepts of reads and writes, because they directly impact the performance and cost of your application.

This topic provides details about the different types of read consistency that apply to DynamoDB. This topic also describes the unit consumption for different read and write operations that you might perform.

Topics

- DynamoDB read consistency
- DynamoDB read and write operations

DynamoDB read consistency

Amazon DynamoDB reads data from tables, local secondary indexes (LSIs), global secondary indexes (GSIs), and streams. For more information, see Core components of Amazon DynamoDB. Both tables and LSIs provide two read consistency options: *eventually consistent* (default) and *strongly consistent* reads. All reads from GSIs and streams are eventually consistent.

When your application writes data to a DynamoDB table and receives an HTTP 200 response (OK), that means the write completed successfully and has been durably persisted. DynamoDB provides *read-committed* isolation and ensures that read operations always return committed values for an item. The read will never present a view to the item from a write which did not ultimately succeed. Read-committed isolation does not prevent modifications of the item immediately after the read operation.

Eventually consistent reads

Eventually consistent is the default read consistent model for all read operations. When issuing eventually consistent reads to a DynamoDB table or an index, the responses may not reflect the results of a recently completed write operation. If you repeat your read request after a short time, the response should eventually return the more recent item. Eventually consistent reads are supported on tables, local secondary indexes, and global secondary indexes. Also note that all reads from a DynamoDB stream are also eventually consistent.

Eventually consistent reads are half the cost of strongly consistent reads. For more information, see Amazon DynamoDB pricing.

Strongly consistent reads

Read operations such as `GetItem`, `Query`, and `Scan` provide an optional `ConsistentRead` parameter. If you set `ConsistentRead` to true, DynamoDB returns a response with the most up-to-date data, reflecting the updates from all prior write operations that were successful. Strongly consistent reads are only supported on tables and local secondary indexes. Strongly consistent reads from a global secondary index or a DynamoDB stream are not supported.

Global tables read consistency

DynamoDB also supports global tables for multi-active and multi-Region replication. A global table is composed of multiple replica tables in different AWS Regions. Any change made to any item in any replica table is replicated to all the other replicas within the same global table, typically within a second, and are eventually consistent. For more information, see Consistency modes.

DynamoDB read and write operations

DynamoDB read operations allow you to retrieve one or more items from a table by specifying the partition key value and, optionally, the sort key value. Using DynamoDB write operations, you can insert, update, or delete items in a table. This topic explains capacity unit consumption for these two operations.

Topics

- Capacity unit consumption for read operations
- Capacity unit consumption for write operations

Capacity unit consumption for read operations

DynamoDB read requests can be either strongly consistent, eventually consistent, or transactional.

- A *strongly consistent* read request of an item up to 4 KB requires one read unit.
- An *eventually consistent* read request of an item up to 4 KB requires one-half read unit.
- A *transactional* read request of an item up to 4 KB requires two read units.

To learn more about DynamoDB read consistency models, see DynamoDB read consistency.

Item sizes for reads are rounded up to the next 4 KB multiple. For example, reading a 3,500-byte item consumes the same throughput as reading a 4 KB item.

If you need to read an item that is larger than 4 KB, DynamoDB needs additional read units. The total number of read units required depends on the item size, and whether you want an eventually consistent or strongly consistent read. For example, if your item size is 8 KB, you require 2 read units to sustain one strongly consistent read. You'll require 1 read unit if you choose eventually consistent reads or 4 read units for a transactional read request.

The following list describes how DynamoDB read operations consume read units:

- GetItem: Reads a single item from a table. To determine the number of read units that `GetItem` will consume, take the item size and round it up to the next 4 KB boundary. This is the number of read units required if you specified a strongly consistent read. For an eventually consistent read, which is the default, divide this number by two.

  For example, if you read an item that is 3.5 KB, DynamoDB rounds the item size to 4 KB. If you read an item of 10 KB, DynamoDB rounds the item size to 12 KB.
- BatchGetItem: Reads up to 100 items from one or more tables. DynamoDB processes each item in the batch as an individual `GetItem` request. DynamoDB first rounds up the size of each item to the next 4 KB boundary and then calculates the total size. The result isn't necessarily the same as the total size of all the items. For example, if `BatchGetItem` reads two items of sizes 1.5 KB and 6.5 KB, DynamoDB calculates the size as 12 KB (4 KB + 8 KB). DynamoDB doesn’t calculate the size as 8 KB (1.5 KB + 6.5 KB).
- Query: Reads multiple items that have the same partition key value. All items returned are treated as a single read operation, where DynamoDB computes the total size of all items. DynamoDB then rounds up the size to the next 4 KB boundary. For example, suppose your query returns 10 items whose combined size is 40.8 KB. DynamoDB rounds the item size for the operation to 44 KB. If a query returns 1500 items of 64 bytes each, the cumulative size is 96 KB.
- Scan: Reads all items in a table. DynamoDB considers the size of the items that are evaluated, not the size of the items returned by the scan. For more information about Scan operations, see Scanning tables in DynamoDB.

> **Important**
>
> If you perform a read operation on an *item that doesn't exist*, DynamoDB will still consume read throughput as outlined above. For `Query`/`Scan` operations, you'll still be charged additional read throughput based on read consistency and the number of partitions searched to serve the request, even if no data exists.

For any operation that returns items, you can request a subset of attributes to retrieve. However, doing so has no impact on the item size calculations. In addition, `Query` and `Scan` can return item counts instead of attribute values. Getting the count of items uses the same quantity of read units and is subject to the same item size calculations. This is because DynamoDB has to read each item in order to increment the count.

Capacity unit consumption for write operations

One write unit represents one write for an item up to 1 KB in size. If you need to write an item that is larger than 1 KB, DynamoDB needs to consume additional write units. Transactional write requests require 2 write units to perform one write for items up to 1 KB. The total number of write request units required depends on the item size. For example, if your item size is 2 KB, you require 2 write units to sustain one write request or 4 write units for a transactional write request.

Item sizes for writes are rounded up to the next 1 KB multiple. For example, writing a 500-byte item consumes the same throughput as writing a 1 KB item.

The following list describes how DynamoDB write operations consume write units:

- PutItem: Writes a single item to a table. If an item with the same primary key exists in the table, the operation replaces the item. For calculating provisioned throughput consumption, the item size that matters is the larger of the two.
- UpdateItem: Modifies a single item in the table. DynamoDB considers the size of the item as it appears before and after the update. The provisioned throughput consumed reflects the larger of these item sizes. Even if you update a subset of the item's attributes, `UpdateItem` will still consume the full amount of provisioned throughput (the larger of the "before" and "after" item sizes).
- DeleteItem: Removes a single item from a table. The provisioned throughput consumption is based on the size of the deleted item.

- BatchWriteItem: Writes up to 25 items to one or more tables. DynamoDB processes each item in the batch as an individual `PutItem` or `DeleteItem` request (updates are not supported). DynamoDB first rounds up the size of each item to the next 1 KB boundary, and then calculates the total size. The result isn't necessarily the same as the total size of all the items. For example, if `BatchWriteItem` writes two items of sizes 500-byte and 3.5 KB, DynamoDB calculates the size as 5 KB (1 KB + 4 KB). DynamoDB doesn't calculate the size as 4 KB (500 bytes + 3.5 KB).

For `PutItem`, `UpdateItem`, and `DeleteItem` operations, DynamoDB rounds the item size up to the next 1 KB. For example, if you put or delete an item of 1.6 KB, DynamoDB rounds the item size up to 2 KB.

`PutItem`, `UpdateItem`, and `DeleteItem` operations allow *conditional writes*, where you specify an expression that must evaluate to true for the operation to succeed. If the expression evaluates to false, DynamoDB still consumes write capacity units from the table. The amount of consumed write capacity units depends on the size of the item. This item can be an existing item in the table or a new one you're attempting to create or update. For example, say that an existing item is 300 KB. The new item you're trying to create or update is 310 KB. The write capacity units consumed will be 310 KB for the new item.

DynamoDB throughput capacity

This section provides an overview of the two throughput modes available for your DynamoDB table and considerations in selecting the appropriate capacity mode for your application. A table's throughput mode determines how the capacity of a table is managed. Throughput mode also determines how you're charged for the read and write operations on your tables. In Amazon DynamoDB, you can choose between **on-demand mode** and **provisioned mode** for your tables to accommodate different workload requirements.

Topics

- On-demand mode
- Provisioned mode
- DynamoDB on-demand capacity mode
- DynamoDB provisioned capacity mode
- Understanding DynamoDB warm throughput
- DynamoDB burst and adaptive capacity
- Considerations when switching capacity modes in DynamoDB

On-demand mode

Amazon DynamoDB on-demand mode is a serverless throughput option that simplifies database management and automatically scales to support customers' most demanding applications. DynamoDB on-demand enables you to create a table without worrying about capacity planning, monitoring usage, and configuring scaling policies. DynamoDB on-demand offers pay-per-request pricing for read and write requests so that you only pay for what you use. For on-demand mode tables, you don't need to specify how much read and write throughput you expect your application to perform.

On-demand mode is the default and recommended throughput option for most DynamoDB workloads. DynamoDB handles all aspects of throughput management and scaling to support workloads that can start small and scale to millions of requests per second. You can read and write to your DynamoDB tables as needed without managing throughput capacity on the table. For more information, see DynamoDB on-demand capacity mode.

Provisioned mode

In provisioned mode, you must specify the number of reads and writes per second that you require for your application. You'll be charged based on the hourly read and write capacity you have provisioned, not how much of that provisioned capacity you actually consumed. This helps you govern your DynamoDB use to stay at or below a defined request rate in order to obtain cost predictability.

You can choose to use provisioned capacity if you have steady workloads with predictable growth, and if you can reliably forecast capacity requirements for your application. For more information, see DynamoDB provisioned capacity mode.

DynamoDB on-demand capacity mode

Amazon DynamoDB on-demand offers a truly serverless database experience that automatically scales to accommodate the most demanding workloads without capacity planning. On-demand simplifies the setup process, eliminates capacity management and monitoring, and provides fast, automatic scaling. With pay-per-request pricing, you don’t have to worry about idle capacity because you only pay for the throughput you actually use. You are billed per read or write request, so your costs directly reflect your actual usage.

When you choose on-demand mode, DynamoDB instantly accommodates your workloads as they ramp up or down to any previously reached traffic level. If a workload’s traffic level hits a new peak, DynamoDB automatically scales to accommodate the increased throughput requirements. On-demand mode is the default and recommended throughput option because it simplifies building modern, serverless applications that can start small and scale to millions of requests per second. Once your on-demand table is scaled out, you can instantly achieve the same throughput again in the future without throttling. If you are driving zero traffic to your table, then with on-demand, you are not charged for any throughput. For more information about on-demand mode's scaling properties, see Initial throughput and scaling properties.

Tables that use on-demand mode deliver the same single-digit millisecond latency, service-level agreement (SLA), and security that DynamoDB provisioned mode offers.

> **Note**
>
> By default, DynamoDB protects you from unintended, runaway usage. To scale beyond the 40,000 table-level read and write throughput limits for all tables in your account, you

can request an increase for this quota. Throughput requests that exceed the default table throughput quota are throttled. For more information, see Throughput default quotas.

Optionally, you can also configure maximum read or write (or both) throughput per second for individual on-demand tables and global secondary indexes. By configuring throughput, you can keep table-level usage and costs bounded, protect against an inadvertent surge in consumed resources, and prevent excessive use for predictable cost management. Throughput requests that exceed the maximum table throughput are throttled. You can modify the table-specific maximum throughput at any time based on your application requirements. For more information, see DynamoDB maximum throughput for on-demand tables.

To get started, create or update a table to use on-demand mode. For more information, see Basic operations on DynamoDB tables.

You can switch tables from provisioned capacity mode to on-demand mode up to four times in a 24-hour rolling window. You can switch tables from on-demand mode to provisioned capacity mode at any time.

For more information about switching between read and write capacity modes, see Considerations when switching capacity modes in DynamoDB. For on-demand table quotas, see Read/write throughput.

Topics

- Read request units and write request units
- Initial throughput and scaling properties
- DynamoDB maximum throughput for on-demand tables

Read request units and write request units

DynamoDB charges you for the reads and writes that your application performs on your tables in terms of *read request units* and *write request units*.

One *read request unit* represents one strongly consistent read operation per second, or two eventually consistent read operations per second, for an item up to 4 KB in size. For more information about DynamoDB read consistency models, see DynamoDB read consistency.

One *write request unit* represents one write operation per second, for an item up to 1 KB in size.

For more information about how read and write units are consumed, see DynamoDB read and write operations.

Initial throughput and scaling properties

DynamoDB tables using on-demand capacity mode automatically adapt to your application's traffic volume. New on-demand tables will be able to sustain up to 4,000 writes per second and 12,000 reads per second. On-demand capacity mode instantly accommodates up to double the previous peak traffic on a table. For example, say that your application's traffic pattern varies between 25,000 and 50,000 strongly consistent reads per second. 50,000 reads per second is the previous traffic peak. On-demand capacity mode instantly accommodates sustained traffic of up to 100,000 reads per second. If your application sustains traffic of 100,000 reads per second, that peak becomes your new previous peak. This previous peak enables subsequent traffic to reach up to 200,000 reads per second.

If your workload generates more than double your previous peak on a table, DynamoDB automatically allocates more capacity as your traffic volume increases. This capacity allocation helps ensure that your workload doesn't experience throttling. However, throttling can occur if you exceed double your previous peak within 30 minutes. For example, say that your application's traffic pattern varies between 25,000 and 50,000 strongly consistent reads per second. 50,000 reads per second is the previously reached traffic peak. We recommend that you either pre-warm your table or space your traffic growth over at least 30 minutes before driving more than 100,000 reads per second. For more information about pre-warming, see Understanding DynamoDB warm throughput.

DynamoDB doesn't place the 30 minute throttling restriction if your workload's peak traffic remains within double your previous peak. If your peak traffic exceeds double the peak, make sure that this growth occurs 30 minutes after you last reached the peak.

DynamoDB maximum throughput for on-demand tables

For on-demand tables, you can optionally specify maximum read or write (or both) throughput per second on individual tables and associated global secondary indexes (GSIs). Specifying a maximum on-demand throughput helps keep table-level usage and costs bounded. By default, maximum throughput settings don't apply and your on-demand throughput rate is bounded by 40,000 table-level read and write throughput AWS service quota for all tables in the account. If needed, you can request an increase to your service quota.

When you configure maximum throughput for an on-demand table, throughput requests that exceed the maximum amount specified will be throttled. You can modify the table-level throughput settings any time based on your application requirements.

The following are some common use cases that can benefit from using maximum throughput for on-demand tables:

- **Throughput cost optimization** – Using maximum throughput for on-demand tables provides an additional layer of cost predictability and manageability. Additionally, it offers greater flexibility to use on-demand mode to support workloads with differing traffic patterns and budget.
- **Protection against excessive usage** – By setting maximum throughput, you can prevent an accidental surge in read or write consumption, which might arise from non-optimized code or rogue processes, against an on-demand table. This table-level setting can protect organizations from consuming excessive resources within a certain time frame.
- **Safeguarding downstream services** – A customer application can include serverless and non-serverless technologies. The serverless piece of the architecture can scale rapidly to match demand. But downstream components with fixed capacities could be overwhelmed. Implementing maximum throughput settings for on-demand tables can prevent large volume of events from propagating to multiple downstream components with unexpected side effects.

You can configure maximum throughput for on-demand mode for new and existing single-Region tables and global tables and GSIs. You can also configure maximum throughput during table restore and data import from Amazon S3 workflows.

You can specify maximum throughput settings for an on-demand tables using the DynamoDB console, AWS CLI, AWS CloudFormation, or DynamoDB API.

> **Note**
>
> The maximum throughput for an on-demand table is applied on a best-effort basis and should be thought of as targets instead of guaranteed request ceilings. Your workload might temporarily exceed the maximum throughput specified because of *burst capacity*. In some cases, DynamoDB uses *burst capacity* to accommodate reads or writes in excess of your table's maximum throughput settings. With burst capacity, unexpected read or write requests can succeed where they otherwise would be throttled.

Topics

- Considerations when using maximum throughput for on-demand mode
- Request throttling and CloudWatch metrics

Considerations when using maximum throughput for on-demand mode

When you use maximum throughput for tables in on-demand mode, the following considerations apply:

- You can independently set maximum throughput for reads and writes for any on-demand table, or individual global secondary index within that table to fine-tune your approach based on specific requirements.
- You can use Amazon CloudWatch to monitor and understand DynamoDB table-level usage metrics and to determine appropriate maximum throughput settings for on-demand mode. For more information, see DynamoDB Metrics and dimensions.
- When you specify the maximum read or write (or both) throughput settings on one global table replica, the same maximum throughput settings are automatically applied to all replica tables. It's important that the replica tables and secondary indexes in a global table have identical write throughput settings to ensure proper replication of data. For more information, see Best practices for global tables.
- The smallest maximum read or write throughput that you can specify is one request unit per second.
- The maximum throughput you specify must be lower than the default throughput quota that is available for any on-demand table, or individual global secondary index within that table.

Request throttling and CloudWatch metrics

If your application exceeds the maximum read or write throughput you've set on your on-demand table, DynamoDB begins to throttle those requests. When DynamoDB throttles a read or write, it returns a `ThrottlingException` to the caller. You can then take appropriate action, if required. For example, you can increase or disable the maximum table throughput setting, or wait for a short interval before retrying the request.

To simplify monitoring the maximum throughput configured for a table or global secondary index, CloudWatch provides the following metrics: OnDemandMaxReadRequestUnits and OnDemandMaxWriteRequestUnits.

DynamoDB provisioned capacity mode

When you create a new provisioned table in DynamoDB, you must specify its *provisioned throughput capacity*. This is the amount of read and write throughput that the table can support. You'll be charged based on the hourly read and write capacity you have provisioned, not how much of that provisioned capacity you actually consumed.

As your application's data and access requirements change, you might need to adjust your table's throughput settings. You can use auto scaling to adjust your table's provisioned capacity automatically in response to traffic changes. DynamoDB auto scaling uses a scaling policy in Application Auto Scaling. To configure auto scaling in DynamoDB, you set the minimum and maximum levels of read and write capacity in addition to the target utilization percentage. Application Auto Scaling creates and manages the CloudWatch alarms that trigger scaling events when the metric deviates from the target. Auto Scaling monitors your table's activity and adjusts its capacity settings up or down based on preconfigured thresholds. Auto scaling triggers when your consumed capacity breaches the configured target utilization for two consecutive minutes. CloudWatch alarms might have a short delay of up to a few minutes before triggering auto scaling. For more information, see Managing throughput capacity automatically with DynamoDB auto scaling.

If you're using DynamoDB auto scaling, the throughput settings are automatically adjusted in response to actual workloads. You can also use the UpdateTable operation to manually adjust your table's throughput capacity. For example, you might decide to do this if you need to bulk-load data from an existing data store into your new DynamoDB table. You could create the table with a large write throughput setting and then reduce this setting after the bulk data load is complete.

> **Note**
>
> By default, DynamoDB protects you from unintended, runaway usage. To scale beyond the 40,000 table-level read and write throughput limits for all tables in your account, you can request an increase for this quota. Throughput requests that exceed the default table throughput quota are throttled. For more information, see Throughput default quotas.

You can switch tables from provisioned capacity mode to on-demand mode up to four times in a 24-hour rolling window. You can switch tables from on-demand mode to provisioned capacity mode at any time.

For more information about switching between read and write capacity modes, see Considerations when switching capacity modes in DynamoDB.

Topics

- Read capacity units and write capacity units
- Choosing initial throughput settings
- DynamoDB auto scaling
- Managing throughput capacity automatically with DynamoDB auto scaling
- DynamoDB reserved capacity

Read capacity units and write capacity units

For provisioned mode tables, you specify throughput requirements in terms of *capacity units*. These units represent the amount of data your application needs to read or write per second. You can modify these settings later, if needed, or enable DynamoDB auto scaling to modify them automatically.

For an item up to 4 KB, one *read capacity unit* (RCU) represents one strongly consistent read operation per second, or two eventually consistent read operations per second. For more information about DynamoDB read consistency models, see DynamoDB read consistency.

A *write capacity unit* (WCU) represents one write per second for an item up to 1 KB. For more information about the different read and write operations, see DynamoDB read and write operations.

Choosing initial throughput settings

Every application has different requirements for reading from and writing to a database. When you're determining the initial throughput settings for a DynamoDB table, consider the following:

- **Expected read and write request rates** — You should estimate the number of reads and writes you need to perform per second.
- **Item sizes** — Some items are small enough that they can be read or written using a single capacity unit. Larger items require multiple capacity units. By estimating the average size of the items that will be in your table, you can specify accurate settings for your table's provisioned throughput.

- **Read consistency requirements** — Read capacity units are based on strongly consistent read operations, which consume twice as many database resources as eventually consistent reads. You should determine whether your application requires strongly consistent reads, or whether it can relax this requirement and perform eventually consistent reads instead. Read operations in DynamoDB are eventually consistent, by default. You can request strongly consistent reads for these operations, if necessary.

For example, say that you want to read 80 items per second from a table. The size of these items is 3 KB, and you want strongly consistent reads. In this case, each read requires one provisioned read capacity unit. To determine this number, divide the item size of the operation by 4 KB. Then, round up to the nearest whole number, as shown in the following example:

- 3 KB / 4 KB = 0.75 or **1** read capacity unit

Therefore, to read 80 items per second from a table, set the table's provisioned read throughput to 80 read capacity units as shown in the following example:

- 1 read capacity unit per item × 80 reads per second = **80** read capacity units

Now suppose that you want to write 100 items per second to your table and that the size of each item is 512 bytes. In this case, each write requires one provisioned write capacity unit. To determine this number, divide the item size of the operation by 1 KB. Then, round up to the nearest whole number, as shown in the following example:

- 512 bytes / 1 KB = 0.5 or **1** write capacity unit

To write 100 items per second to your table, set the table's provisioned write throughput to 100 write capacity units:

- 1 write capacity unit per item × 100 writes per second = **100** write capacity units

DynamoDB auto scaling

DynamoDB auto scaling actively manages provisioned throughput capacity for tables and global secondary indexes. With auto scaling, you define a range (upper and lower limits) for read and write capacity units. You also define a target utilization percentage within that range. DynamoDB

auto scaling seeks to maintain your target utilization, even as your application workload increases or decreases.

With DynamoDB auto scaling, a table or a global secondary index can increase its provisioned read and write capacity to handle sudden increases in traffic, without request throttling. When the workload decreases, DynamoDB auto scaling can decrease the throughput so that you don't pay for unused provisioned capacity.

> **Note**
>
> If you use the AWS Management Console to create a table or a global secondary index, DynamoDB auto scaling is enabled by default.
> You can manage auto scaling settings at any time by using the console, the AWS CLI, or one of the AWS SDKs. For more information, see Managing throughput capacity automatically with DynamoDB auto scaling.

Utilization rate

Utilization rate can help you determine if you're over provisioning capacity, in which case should reduce your table capacity to save costs. Conversely, it can also help you determine if you're under provisioning capacity. In this case, you should increase table capacity to prevent potential throttling of requests during unexpected high traffic instances. For more information, see Amazon DynamoDB auto scaling: Performance and cost optimization at any scale.

If you're using DynamoDB auto scaling, you'll also need to set a target utilization percentage. Auto scaling will use this percentage as a target to adjust capacity upward or downward. We recommend setting target utilization to 70%. For more information, see Managing throughput capacity automatically with DynamoDB auto scaling.

Managing throughput capacity automatically with DynamoDB auto scaling

Many database workloads are cyclical in nature, while others are difficult to predict in advance. For one example, consider a social networking app where most of the users are active during daytime hours. The database must be able to handle the daytime activity, but there's no need for the same levels of throughput at night. For another example, consider a new mobile gaming app that is experiencing unexpectedly rapid adoption. If the game becomes too popular it could exceed

the available database resources, resulting in slow performance and unhappy customers. These kinds of workloads often require manual intervention to scale database resources up or down in response to varying usage levels.

Amazon DynamoDB auto scaling uses the AWS Application Auto Scaling service to dynamically adjust provisioned throughput capacity on your behalf, in response to actual traffic patterns. This enables a table or a global secondary index (GSI) to increase its provisioned read and write capacity to handle sudden increases in traffic, without throttling. When the workload decreases, Application Auto Scaling decreases the throughput so that you don't pay for unused provisioned capacity.

> **Note**
>
> If you use the AWS Management Console to create a table or a global secondary index, DynamoDB auto scaling is enabled by default. You can modify your auto scaling settings at any time. For more information, see Using the AWS Management Console with DynamoDB auto scaling.
>
> When you delete a table or global table replica then any associated scalable targets, scaling polices, or CloudWatch alarms are not automatically deleted with it.

With Application Auto Scaling, you create a *scaling policy* for a table or a global secondary index. The scaling policy specifies whether you want to scale read capacity or write capacity (or both), and the minimum and maximum provisioned capacity unit settings for the table or index.

The scaling policy also contains a *target utilization*—the percentage of consumed provisioned throughput at a point in time. Application Auto Scaling uses a *target tracking* algorithm to adjust the provisioned throughput of the table (or index) upward or downward in response to actual workloads, so that the actual capacity utilization remains at or near your target utilization.

DynamoDB outputs consumed provisioned throughput for one-minute periods. Auto scaling triggers when your consumed capacity breaches the configured target utilization for two consecutive minutes. CloudWatch alarms might have a short delay of up to a few minutes before triggering auto scaling. This delay ensures accurate CloudWatch metric evaluation. If the consumed throughput spikes are more than a minute apart, auto scaling might not trigger. Similarly, a scale down event can occur when 15 consecutive data points are lower than the target utilization.
In either case, after auto scaling triggers, the UpdateTable API is invoked. It then takes several minutes to update the provisioned capacity for the table or index. During this period, any requests that exceed the previous provisioned capacity of the tables are throttled.

Important

You can't adjust the number of data points to breach to trigger the underlying alarm (though the current number could change in the future).

You can set the auto scaling target utilization values between 20 and 90 percent for your read and write capacity.

Note

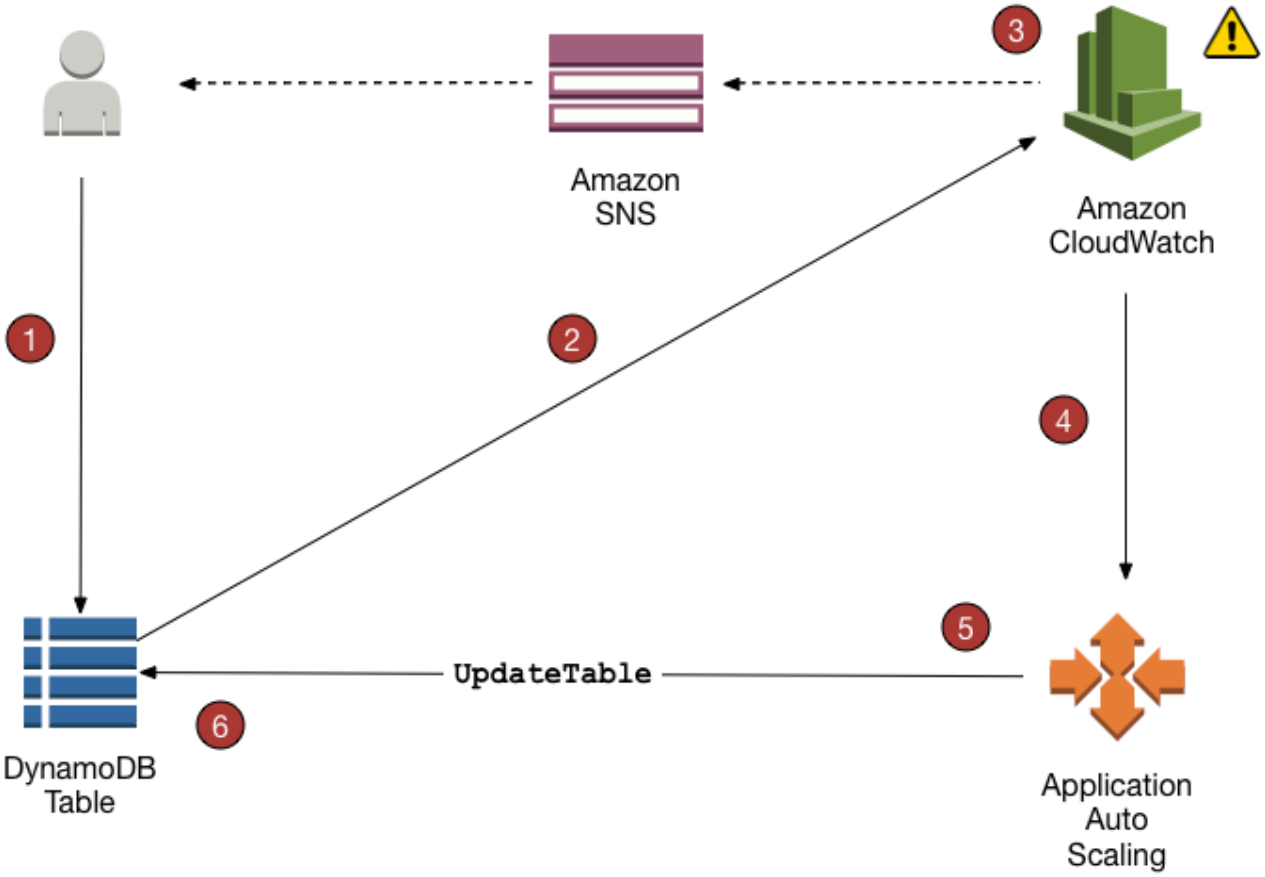
In addition to tables, DynamoDB auto scaling also supports global secondary indexes. Every global secondary index has its own provisioned throughput capacity, separate from that of its base table. When you create a scaling policy for a global secondary index, Application Auto Scaling adjusts the provisioned throughput settings for the index to ensure that its actual utilization stays at or near your desired utilization ratio.

How DynamoDB auto scaling works

Note

To get started quickly with DynamoDB auto scaling, see Using the AWS Management Console with DynamoDB auto scaling.

The following diagram provides a high-level overview of how DynamoDB auto scaling manages throughput capacity for a table.



The following steps summarize the auto scaling process as shown in the previous diagram:

1. You create an Application Auto Scaling policy for your DynamoDB table.
2. DynamoDB publishes consumed capacity metrics to Amazon CloudWatch.
3. If the table's consumed capacity exceeds your target utilization (or falls below the target) for a specific length of time, Amazon CloudWatch triggers an alarm. You can view the alarm on the console and receive notifications using Amazon Simple Notification Service (Amazon SNS).
4. The CloudWatch alarm invokes Application Auto Scaling to evaluate your scaling policy.
5. Application Auto Scaling issues an `UpdateTable` request to adjust your table's provisioned throughput.
6. DynamoDB processes the `UpdateTable` request, dynamically increasing (or decreasing) the table's provisioned throughput capacity so that it approaches your target utilization.

To understand how DynamoDB auto scaling works, suppose that you have a table named `ProductCatalog`. The table is bulk-loaded with data infrequently, so it doesn't incur very much

write activity. However, it does experience a high degree of read activity, which varies over time. By monitoring the Amazon CloudWatch metrics for `ProductCatalog`, you determine that the table requires 1,200 read capacity units (to avoid DynamoDB throttling read requests when activity is at its peak). You also determine that `ProductCatalog` requires 150 read capacity units at a minimum, when read traffic is at its lowest point. For more information about preventing throttling, see Troubleshooting throttling in Amazon DynamoDB.

Within the range of 150 to 1,200 read capacity units, you decide that a target utilization of 70 percent would be appropriate for the `ProductCatalog` table. *Target utilization* is the ratio of consumed capacity units to provisioned capacity units, expressed as a percentage. Application Auto Scaling uses its target tracking algorithm to ensure that the provisioned read capacity of `ProductCatalog` is adjusted as required so that utilization remains at or near 70 percent.

> **Note**
>
> DynamoDB auto scaling modifies provisioned throughput settings only when the actual workload stays elevated or depressed for a sustained period of several minutes. The Application Auto Scaling target tracking algorithm seeks to keep the target utilization at or near your chosen value over the long term.
> Sudden, short-duration spikes of activity are accommodated by the table's built-in burst capacity. For more information, see Burst capacity.

To enable DynamoDB auto scaling for the `ProductCatalog` table, you create a scaling policy. This policy specifies the following:

- The table or global secondary index that you want to manage
- Which capacity type to manage (read capacity or write capacity)
- The upper and lower boundaries for the provisioned throughput settings
- Your target utilization

When you create a scaling policy, Application Auto Scaling creates a pair of Amazon CloudWatch alarms on your behalf. Each pair represents the upper and lower boundaries for your provisioned throughput settings. These CloudWatch alarms are triggered when the table's actual utilization deviates from your target utilization for a sustained period of time.

When one of the CloudWatch alarms is triggered, Amazon SNS sends you a notification (if you have enabled it). The CloudWatch alarm then invokes Application Auto Scaling, which in turn notifies DynamoDB to adjust the `ProductCatalog` table's provisioned capacity upward or downward as appropriate.

During a scaling event, AWS Config is charged per configuration item recorded. When a scaling event occurs, four CloudWatch alarms are created for each read and write auto-scaling event: ProvisionedCapacity alarms: ProvisionedCapacityLow, ProvisionedCapacityHigh and ConsumedCapacity alarms: AlarmHigh, AlarmLow. This results in a total of eight alarms. Therefore, AWS Config records eight configuration items for every scaling event.

> **Note**
>
> You can also schedule your DynamoDB scaling so it happens at certain times. Learn the basic steps here.

Usage notes

Before you begin using DynamoDB auto scaling, you should be aware of the following:

- DynamoDB auto scaling can increase read capacity or write capacity as often as necessary, in accordance with your auto scaling policy. All DynamoDB quotas remain in effect, as described in Quotas in Amazon DynamoDB.
- DynamoDB auto scaling doesn't prevent you from manually modifying provisioned throughput settings. These manual adjustments don't affect any existing CloudWatch alarms that are related to DynamoDB auto scaling.
- If you enable DynamoDB auto scaling for a table that has one or more global secondary indexes, we highly recommend that you also apply auto scaling uniformly to those indexes. This will help ensure better performance for table writes and reads, and help avoid throttling. You can enable auto scaling by selecting **Apply same settings to global secondary indexes** in the AWS Management Console. For more information, see Enabling DynamoDB auto scaling on existing tables.
- When you delete a table or global table replica, any associated scalable targets, scaling polices or CloudWatch alarms are not automatically deleted with it.

- When creating a GSI for an existing table, auto scaling is not enabled for the GSI. You will have to manually manage the capacity while the GSI is being built. Once the backfill on the GSI completes and it reaches active status, auto scaling will operate as normal.

Using the AWS Management Console with DynamoDB auto scaling

When you use the AWS Management Console to create a new table, Amazon DynamoDB auto scaling is enabled for that table by default. You can also use the console to enable auto scaling for existing tables, modify auto scaling settings, or disable auto scaling.

> **Note**
>
> For more advanced features like setting scale-in and scale-out cooldown times, use the AWS Command Line Interface (AWS CLI) to manage DynamoDB auto scaling. For more information, see Using the AWS CLI to manage DynamoDB auto scaling.

Topics

- Before you begin: Granting user permissions for DynamoDB auto scaling
- Creating a new table with auto scaling enabled
- Enabling DynamoDB auto scaling on existing tables
- Viewing auto scaling activities on the console
- Modifying or disabling DynamoDB auto scaling settings

Before you begin: Granting user permissions for DynamoDB auto scaling

In AWS Identity and Access Management (IAM), the AWS managed policy `DynamoDBFullAccess` provides the required permissions for using the DynamoDB console. However, for DynamoDB auto scaling, users require additional permissions.

> **Important**
>
> To delete an auto scaling-enabled table, `application-autoscaling:*` permissions are required. The AWS managed policy `DynamoDBFullAccess` includes such permissions.

To set up a user for DynamoDB console access and DynamoDB auto scaling, create a role and add the **AmazonDynamoDBFullAccess** policy to that role. Then assign the role to a user.

Creating a new table with auto scaling enabled

> **Note**
>
> DynamoDB auto scaling requires the presence of a service-linked role (`AWSServiceRoleForApplicationAutoScaling_DynamoDBTable`) that performs auto scaling actions on your behalf. This role is created automatically for you. For more information, see Service-linked roles for Application Auto Scaling in the*Application Auto Scaling User Guide*.

To create a new table with auto scaling enabled

1. Open the DynamoDB console at https://console.aws.amazon.com/dynamodb/.
2. Choose **Create table**.
3. On the **Create table** page, enter the **Table name** and primary key details.
4. If you choose **Default settings**, auto scaling is enabled in the new table.

   Otherwise, choose **Customize settings** and do the following to specify custom settings for the table:

   a. For **Table class**, keep the default selection of **DynamoDB Standard**.

   b. For **Read/write capacity settings**, keep the default selection of **Provisioned**, then do the following:

      i. For **Read capacity**, make sure **Auto scaling** is set to **On**.

      ii. For **Write capacity**, make sure **Auto scaling** is set to **On**.

      iii. For **Read capacity** and **Write capacity**, set your desired scaling policy for the table and, optionally, all global secondary indexes of the table.

      - **Minimum capacity units** – Enter your lower boundary for the auto scaling range.
      - **Maximum capacity units** – Enter your upper boundary for the auto scaling range.
      - **Target utilization** – Enter your target utilization percentage for the table.

> **Note**
>
> If you create a global secondary index for the new table, the index's capacity at time of creation will be the same as your base table's capacity. You can change the index's capacity in the table's settings after you create the table.

5. Choose **Create table**. This creates your table with the auto scaling parameters you specified.

Enabling DynamoDB auto scaling on existing tables

> **Note**
>
> DynamoDB auto scaling requires the presence of a service-linked role (`AWSServiceRoleForApplicationAutoScaling_DynamoDBTable`) that performs auto scaling actions on your behalf. This role is created automatically for you. For more information, see Service-linked roles for Application Auto Scaling.

To enable DynamoDB auto scaling for an existing table

1. Open the DynamoDB console at https://console.aws.amazon.com/dynamodb/.
2. In the navigation pane on the left side of the console, choose **Tables**.
3. Choose the table on which you want to enable auto scaling, and then do the following:
   a. Choose the **Additional settings** tab.
   b. In the **Read/write capacity** section, choose **Edit**.
   c. In the **Capacity mode** section, choose **Provisioned**.
   d. In the **Table capacity** section, set **Auto scaling** to **On** for **Read capacity**, **Write capacity**, or both. For each of these, set your desired scaling policy for the table and, optionally, all global secondary indexes of the table.
      - **Minimum capacity units** – Enter your lower boundary for the auto scaling range.
      - **Maximum capacity units** – Enter your upper boundary for the auto scaling range.
      - **Target utilization** – Enter your target utilization percentage for the table.

- **Use the same capacity read/write capacity settings for all global secondary indexes** – Choose whether global secondary indexes should use the same auto scaling policy as the base table.

  > **Note**
  >
  > For best performance, we recommend that you enable **Use the same read/write capacity settings for all global secondary indexes**. This option allows DynamoDB auto scaling to uniformly scale all the global secondary indexes on the base table. This includes existing global secondary indexes, and any others that you create for this table in the future.
  > With this option enabled, you can't set a scaling policy on an individual global secondary index.

4. When the settings are as you want them, choose **Save**.

Viewing auto scaling activities on the console

As your application drives read and write traffic to your table, DynamoDB auto scaling dynamically modifies the table's throughput settings. Amazon CloudWatch keeps track of provisioned and consumed capacity, throttled events, latency, and other metrics for all of your DynamoDB tables and secondary indexes.

To view these metrics in the DynamoDB console, choose the table that you want to work with and choose the **Monitor** tab. To create a customizable view of table metrics, select **View all in CloudWatch**.

Modifying or disabling DynamoDB auto scaling settings

You can use the AWS Management Console to modify your DynamoDB auto scaling settings. To do this, go to the **Additional settings** tab for your table, and choose **Edit** in the **Read/write capacity** section. For more information about these settings, see Enabling DynamoDB auto scaling on existing tables.

Using the AWS CLI to manage DynamoDB auto scaling

Instead of using the AWS Management Console, you can use the AWS Command Line Interface (AWS CLI) to manage Amazon DynamoDB auto scaling. The tutorial in this section demonstrates

how to install and configure the AWS CLI for managing DynamoDB auto scaling. In this tutorial, you do the following:

- Create a DynamoDB table named `TestTable`. The initial throughput settings are 5 read capacity units and 5 write capacity units.
- Create an Application Auto Scaling policy for `TestTable`. The policy seeks to maintain a 50 percent target ratio between consumed write capacity and provisioned write capacity. The range for this metric is between 5 and 10 write capacity units. (Application Auto Scaling is not allowed to adjust the throughput beyond this range.)
- Run a Python program to drive write traffic to `TestTable`. When the target ratio exceeds 50 percent for a sustained period of time, Application Auto Scaling notifies DynamoDB to adjust the throughput of `TestTable` upward to maintain the 50 percent target utilization.
- Verify that DynamoDB has successfully adjusted the provisioned write capacity for `TestTable`.

> **Note**
>
> You can also schedule your DynamoDB scaling so it happens at certain times. Learn the basic steps here.

Topics

- Before you begin
- Step 1: Create a DynamoDB table
- Step 2: Register a scalable target
- Step 3: Create a scaling policy
- Step 4: Drive write traffic to TestTable
- Step 5: View Application Auto Scaling actions
- (Optional) Step 6: Clean up

Before you begin

Complete the following tasks before starting the tutorial.

Install the AWS CLI

If you haven't already done so, you must install and configure the AWS CLI. To do this, follow these instructions in the *AWS Command Line Interface User Guide*:

- Installing the AWS CLI
- Configuring the AWS CLI

Install Python

Part of this tutorial requires you to run a Python program (see Step 4: Drive write traffic to TestTable). If you don't already have it installed, you can download Python.

Step 1: Create a DynamoDB table

In this step, you use the AWS CLI to create `TestTable`. The primary key consists of `pk` (partition key) and `sk` (sort key). Both of these attributes are of type `Number`. The initial throughput settings are 5 read capacity units and 5 write capacity units.

1. Use the following AWS CLI command to create the table.

```
aws dynamodb create-table \
    --table-name TestTable \
    --attribute-definitions \
        AttributeName=pk,AttributeType=N \
        AttributeName=sk,AttributeType=N \
    --key-schema \
        AttributeName=pk,KeyType=HASH \
        AttributeName=sk,KeyType=RANGE \
    --provisioned-throughput ReadCapacityUnits=5,WriteCapacityUnits=5
```

2. To check the status of the table, use the following command.

```
aws dynamodb describe-table \
    --table-name TestTable \
    --query "Table.[TableName,TableStatus,ProvisionedThroughput]"
```

   The table is ready for use when its status is `ACTIVE`.

Step 2: Register a scalable target

Next you register the table's write capacity as a scalable target with Application Auto Scaling. This allows Application Auto Scaling to adjust the provisioned write capacity for *TestTable*, but only within the range of 5–10 capacity units.

> **Note**
>
> DynamoDB auto scaling requires the presence of a service linked role (`AWSServiceRoleForApplicationAutoScaling_DynamoDBTable`) that performs auto scaling actions on your behalf. This role is created automatically for you. For more information, see Service-linked roles for Application Auto Scaling in the *Application Auto Scaling User Guide*.

1. Enter the following command to register the scalable target.

```
aws application-autoscaling register-scalable-target \
    --service-namespace dynamodb \
    --resource-id "table/TestTable" \
    --scalable-dimension "dynamodb:table:WriteCapacityUnits" \
    --min-capacity 5 \
    --max-capacity 10
```

2. To verify the registration, use the following command.

```
aws application-autoscaling describe-scalable-targets \
    --service-namespace dynamodb \
    --resource-id "table/TestTable"
```

> **Note**
>
> You can also register a scalable target against a global secondary index. For example, for a global secondary index ("test-index"), the resource ID and scalable dimension arguments are updated appropriately.
>
> ```
> aws application-autoscaling register-scalable-target \
>     --service-namespace dynamodb \
>     --resource-id "table/TestTable/index/test-index" \
>     --scalable-dimension "dynamodb:index:WriteCapacityUnits" \
> ```

```
    --min-capacity 5 \
    --max-capacity 10
```

Step 3: Create a scaling policy

In this step, you create a scaling policy for `TestTable`. The policy defines the details under which Application Auto Scaling can adjust your table's provisioned throughput, and what actions to take when it does so. You associate this policy with the scalable target that you defined in the previous step (write capacity units for the `TestTable` table).

The policy contains the following elements:

- `PredefinedMetricSpecification`—The metric that Application Auto Scaling is allowed to adjust. For DynamoDB, the following values are valid values for `PredefinedMetricType`:
  - `DynamoDBReadCapacityUtilization`
  - `DynamoDBWriteCapacityUtilization`
- `ScaleOutCooldown`—The minimum amount of time (in seconds) between each Application Auto Scaling event that increases provisioned throughput. This parameter allows Application Auto Scaling to continuously, but not aggressively, increase the throughput in response to real-world workloads. The default setting for `ScaleOutCooldown` is 0.
- `ScaleInCooldown`—The minimum amount of time (in seconds) between each Application Auto Scaling event that decreases provisioned throughput. This parameter allows Application Auto Scaling to decrease the throughput gradually and predictably. The default setting for `ScaleInCooldown` is 0.
- `TargetValue`—Application Auto Scaling ensures that the ratio of consumed capacity to provisioned capacity stays at or near this value. You define `TargetValue` as a percentage.

Note

To further understand how `TargetValue` works, suppose that you have a table with a provisioned throughput setting of 200 write capacity units. You decide to create a scaling policy for this table, with a `TargetValue` of 70 percent.
Now suppose that you begin driving write traffic to the table so that the actual write throughput is 150 capacity units. The consumed-to-provisioned ratio is now (150 / 200), or 75 percent. This ratio exceeds your target, so Application Auto Scaling increases the

provisioned write capacity to 215 so that the ratio is (150 / 215), or 69.77 percent—as close to your `TargetValue` as possible, but not exceeding it.

For `TestTable`, you set `TargetValue` to 50 percent. Application Auto Scaling adjusts the table's provisioned throughput within the range of 5–10 capacity units (see Step 2: Register a scalable target) so that the consumed-to-provisioned ratio remains at or near 50 percent. You set the values for `ScaleOutCooldown` and `ScaleInCooldown` to 60 seconds.

1. Create a file named `scaling-policy.json` with the following contents.

```
{
    "PredefinedMetricSpecification": {
        "PredefinedMetricType": "DynamoDBWriteCapacityUtilization"
    },
    "ScaleOutCooldown": 60,
    "ScaleInCooldown": 60,
    "TargetValue": 50.0
}
```

2. Use the following AWS CLI command to create the policy.

```
aws application-autoscaling put-scaling-policy \
    --service-namespace dynamodb \
    --resource-id "table/TestTable" \
    --scalable-dimension "dynamodb:table:WriteCapacityUnits" \
    --policy-name "MyScalingPolicy" \
    --policy-type "TargetTrackingScaling" \
    --target-tracking-scaling-policy-configuration file://scaling-policy.json
```

3. In the output, note that Application Auto Scaling has created two Amazon CloudWatch alarms—one each for the upper and lower boundary of the scaling target range.
4. Use the following AWS CLI command to view more details about the scaling policy.

```
aws application-autoscaling describe-scaling-policies \
    --service-namespace dynamodb \
    --resource-id "table/TestTable" \
    --policy-name "MyScalingPolicy"
```

5. In the output, verify that the policy settings match your specifications from Step 2: Register a scalable target and Step 3: Create a scaling policy.

Step 4: Drive write traffic to TestTable

Now you can test your scaling policy by writing data to `TestTable`. To do this, you run a Python program.

1. Create a file named `bulk-load-test-table.py` with the following contents.

```
import boto3
dynamodb = boto3.resource('dynamodb')

table = dynamodb.Table("TestTable")

filler = "x" * 100000

i = 0
while (i < 10):
    j = 0
    while (j < 10):
        print (i, j)

        table.put_item(
            Item={
                'pk':i,
                'sk':j,
                'filler':{"S":filler}
            }
        )
        j += 1
    i += 1
```

2. Enter the following command to run the program.

   ```
   python bulk-load-test-table.py
   ```

   The provisioned write capacity for `TestTable` is very low (5 write capacity units), so the program stalls occasionally due to write throttling. This is expected behavior.

   Let the program continue running while you move on to the next step.

Step 5: View Application Auto Scaling actions

In this step, you view the Application Auto Scaling actions that are initiated on your behalf. You also verify that Application Auto Scaling has updated the provisioned write capacity for `TestTable`.

1. Enter the following command to view the Application Auto Scaling actions.

```
aws application-autoscaling describe-scaling-activities \
    --service-namespace dynamodb
```

   Rerun this command occasionally, while the Python program is running. (It takes several minutes before your scaling policy is invoked.) You should eventually see the following output.

```
...
{
    "ScalableDimension": "dynamodb:table:WriteCapacityUnits",
    "Description": "Setting write capacity units to 10.",
    "ResourceId": "table/TestTable",
    "ActivityId": "0cc6fb03-2a7c-4b51-b67f-217224c6b656",
    "StartTime": 1489088210.175,
    "ServiceNamespace": "dynamodb",
    "EndTime": 1489088246.85,
    "Cause": "monitor alarm AutoScaling-table/TestTable-
AlarmHigh-1bb3c8db-1b97-4353-baf1-4def76f4e1b9 in state ALARM triggered policy
 MyScalingPolicy",
    "StatusMessage": "Successfully set write capacity units to 10. Change
 successfully fulfilled by dynamodb.",
    "StatusCode": "Successful"
},
...
```

   This indicates that Application Auto Scaling has issued an `UpdateTable` request to DynamoDB.

2. Enter the following command to verify that DynamoDB increased the table's write capacity.

```
aws dynamodb describe-table \
    --table-name TestTable \
    --query "Table.[TableName,TableStatus,ProvisionedThroughput]"
```

The `WriteCapacityUnits` should have been scaled from `5` to `10`.

(Optional) Step 6: Clean up

In this tutorial, you created several resources. You can delete these resources if you no longer need them.

1. Delete the scaling policy for `TestTable`.

```
aws application-autoscaling delete-scaling-policy \
    --service-namespace dynamodb \
    --resource-id "table/TestTable" \
    --scalable-dimension "dynamodb:table:WriteCapacityUnits" \
    --policy-name "MyScalingPolicy"
```

2. Deregister the scalable target.

```
aws application-autoscaling deregister-scalable-target \
    --service-namespace dynamodb \
    --resource-id "table/TestTable" \
    --scalable-dimension "dynamodb:table:WriteCapacityUnits"
```

3. Delete the `TestTable` table.

```
aws dynamodb delete-table --table-name TestTable
```

Using the AWS SDK to configure auto scaling on Amazon DynamoDB tables

In addition to using the AWS Management Console and the AWS Command Line Interface (AWS CLI), you can write applications that interact with Amazon DynamoDB auto scaling. This section contains two Java programs that you can use to test this functionality:

- `EnableDynamoDBAutoscaling.java`
- `DisableDynamoDBAutoscaling.java`

Enabling Application Auto Scaling for a table

The following program shows an example of setting up an auto scaling policy for a DynamoDB table (`TestTable`). It proceeds as follows:

- The program registers write capacity units as a scalable target for `TestTable`. The range for this metric is between 5 and 10 write capacity units.
- After the scalable target is created, the program builds a target tracking configuration. The policy seeks to maintain a 50 percent target ratio between consumed write capacity and provisioned write capacity.
- The program then creates the scaling policy, based on the target tracking configuration.

> **Note**
>
> When you manually remove a table or global table replica, you do not automatically remove any associated scalable targets, scaling policies, or CloudWatch alarms.

Java v2

```
import software.amazon.awssdk.regions.Region;
import
 software.amazon.awssdk.services.applicationautoscaling.ApplicationAutoScalingClient;
import
 software.amazon.awssdk.services.applicationautoscaling.model.ApplicationAutoScalingExceptio
import
 software.amazon.awssdk.services.applicationautoscaling.model.DescribeScalableTargetsRequest
import
 software.amazon.awssdk.services.applicationautoscaling.model.DescribeScalableTargetsRespons
import
 software.amazon.awssdk.services.applicationautoscaling.model.DescribeScalingPoliciesRequest
import
 software.amazon.awssdk.services.applicationautoscaling.model.DescribeScalingPoliciesRespons
import software.amazon.awssdk.services.applicationautoscaling.model.PolicyType;
import
 software.amazon.awssdk.services.applicationautoscaling.model.PredefinedMetricSpecification;
import
 software.amazon.awssdk.services.applicationautoscaling.model.PutScalingPolicyRequest;
import
 software.amazon.awssdk.services.applicationautoscaling.model.RegisterScalableTargetRequest;
```

```
import software.amazon.awssdk.services.applicationautoscaling.model.ScalingPolicy;
import
 software.amazon.awssdk.services.applicationautoscaling.model.ServiceNamespace;
import
 software.amazon.awssdk.services.applicationautoscaling.model.ScalableDimension;
import software.amazon.awssdk.services.applicationautoscaling.model.MetricType;
import
 software.amazon.awssdk.services.applicationautoscaling.model.TargetTrackingScalingPolicyCon
import java.util.List;

/**
 * Before running this Java V2 code example, set up your development environment,
 including your credentials.
 *
 * For more information, see the following documentation topic:
 *
 * https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-started.html
 */
public class EnableDynamoDBAutoscaling {
    public static void main(String[] args) {
        final String usage = """

            Usage:
               <tableId> <roleARN> <policyName>\s

            Where:
               tableId - The table Id value (for example, table/Music).
               roleARN - The ARN of the role that has ApplicationAutoScaling
 permissions.
               policyName - The name of the policy to create.

            """;

        if (args.length != 3) {
            System.out.println(usage);
            System.exit(1);
        }

        System.out.println("This example registers an Amazon DynamoDB table, which
 is the resource to scale.");
        String tableId = args[0];
        String roleARN = args[1];
        String policyName = args[2];
        ServiceNamespace ns = ServiceNamespace.DYNAMODB;
```

```
       ScalableDimension tableWCUs =
ScalableDimension.DYNAMODB_TABLE_WRITE_CAPACITY_UNITS;
       ApplicationAutoScalingClient appAutoScalingClient =
ApplicationAutoScalingClient.builder()
           .region(Region.US_EAST_1)
           .build();

       registerScalableTarget(appAutoScalingClient, tableId, roleARN, ns,
tableWCUs);
       verifyTarget(appAutoScalingClient, tableId, ns, tableWCUs);
       configureScalingPolicy(appAutoScalingClient, tableId, ns, tableWCUs,
policyName);
   }

   public static void registerScalableTarget(ApplicationAutoScalingClient
appAutoScalingClient, String tableId, String roleARN, ServiceNamespace ns,
ScalableDimension tableWCUs) {
       try {
           RegisterScalableTargetRequest targetRequest =
RegisterScalableTargetRequest.builder()
               .serviceNamespace(ns)
               .scalableDimension(tableWCUs)
               .resourceId(tableId)
               .roleARN(roleARN)
               .minCapacity(5)
               .maxCapacity(10)
               .build();

           appAutoScalingClient.registerScalableTarget(targetRequest);
           System.out.println("You have registered " + tableId);

       } catch (ApplicationAutoScalingException e) {
           System.err.println(e.awsErrorDetails().errorMessage());
       }
   }

   // Verify that the target was created.
   public static void verifyTarget(ApplicationAutoScalingClient
appAutoScalingClient, String tableId, ServiceNamespace ns, ScalableDimension
tableWCUs) {
       DescribeScalableTargetsRequest dscRequest =
DescribeScalableTargetsRequest.builder()
           .scalableDimension(tableWCUs)
           .serviceNamespace(ns)
```

```
            .resourceIds(tableId)
            .build();

        DescribeScalableTargetsResponse response =
appAutoScalingClient.describeScalableTargets(dscRequest);
        System.out.println("DescribeScalableTargets result: ");
        System.out.println(response);
    }

    // Configure a scaling policy.
    public static void configureScalingPolicy(ApplicationAutoScalingClient
appAutoScalingClient, String tableId, ServiceNamespace ns, ScalableDimension
tableWCUs, String policyName) {
        // Check if the policy exists before creating a new one.
        DescribeScalingPoliciesResponse describeScalingPoliciesResponse =
appAutoScalingClient.describeScalingPolicies(DescribeScalingPoliciesRequest.builder()
            .serviceNamespace(ns)
            .resourceId(tableId)
            .scalableDimension(tableWCUs)
            .build());

        if (!describeScalingPoliciesResponse.scalingPolicies().isEmpty()) {
            // If policies exist, consider updating an existing policy instead of
creating a new one.
            System.out.println("Policy already exists. Consider updating it
instead.");
            List<ScalingPolicy> polList =
describeScalingPoliciesResponse.scalingPolicies();
            for (ScalingPolicy pol : polList) {
                System.out.println("Policy name:" +pol.policyName());
            }
        } else {
            // If no policies exist, proceed with creating a new policy.
            PredefinedMetricSpecification specification =
PredefinedMetricSpecification.builder()

.predefinedMetricType(MetricType.DYNAMO_DB_WRITE_CAPACITY_UTILIZATION)
                .build();

            TargetTrackingScalingPolicyConfiguration policyConfiguration =
TargetTrackingScalingPolicyConfiguration.builder()
                .predefinedMetricSpecification(specification)
                .targetValue(50.0)
                .scaleInCooldown(60)
```

```
                .scaleOutCooldown(60)
                .build();

            PutScalingPolicyRequest putScalingPolicyRequest =
 PutScalingPolicyRequest.builder()
                .targetTrackingScalingPolicyConfiguration(policyConfiguration)
                .serviceNamespace(ns)
                .scalableDimension(tableWCUs)
                .resourceId(tableId)
                .policyName(policyName)
                .policyType(PolicyType.TARGET_TRACKING_SCALING)
                .build();

            try {
                appAutoScalingClient.putScalingPolicy(putScalingPolicyRequest);
                System.out.println("You have successfully created a scaling policy
 for an Application Auto Scaling scalable target");
            } catch (ApplicationAutoScalingException e) {
                System.err.println("Error: " + e.awsErrorDetails().errorMessage());
            }
        }
    }
}
```

Java v1

The program requires that you provide an Amazon Resource Name (ARN) for a valid Application Auto Scaling service linked role. (For example: `arn:aws:iam::122517410325:role/AWSServiceRoleForApplicationAutoScaling_DynamoDBTable`.) In the following program, replace `SERVICE_ROLE_ARN_GOES_HERE` with the actual ARN.

```
package com.amazonaws.codesamples.autoscaling;

import
 com.amazonaws.services.applicationautoscaling.AWSApplicationAutoScalingClient;
import
 com.amazonaws.services.applicationautoscaling.AWSApplicationAutoScalingClientBuilder;
import
 com.amazonaws.services.applicationautoscaling.model.DescribeScalableTargetsRequest;
import
 com.amazonaws.services.applicationautoscaling.model.DescribeScalableTargetsResult;
import
 com.amazonaws.services.applicationautoscaling.model.DescribeScalingPoliciesRequest;
```

```
import
 com.amazonaws.services.applicationautoscaling.model.DescribeScalingPoliciesResult;
import com.amazonaws.services.applicationautoscaling.model.MetricType;
import com.amazonaws.services.applicationautoscaling.model.PolicyType;
import
 com.amazonaws.services.applicationautoscaling.model.PredefinedMetricSpecification;
import com.amazonaws.services.applicationautoscaling.model.PutScalingPolicyRequest;
import
 com.amazonaws.services.applicationautoscaling.model.RegisterScalableTargetRequest;
import com.amazonaws.services.applicationautoscaling.model.ScalableDimension;
import com.amazonaws.services.applicationautoscaling.model.ServiceNamespace;
import
 com.amazonaws.services.applicationautoscaling.model.TargetTrackingScalingPolicyConfiguratio

public class EnableDynamoDBAutoscaling {

 static AWSApplicationAutoScalingClient aaClient = (AWSApplicationAutoScalingClient)
 AWSApplicationAutoScalingClientBuilder
   .standard().build();

 public static void main(String args[]) {

  ServiceNamespace ns = ServiceNamespace.Dynamodb;
  ScalableDimension tableWCUs = ScalableDimension.DynamodbTableWriteCapacityUnits;
  String resourceID = "table/TestTable";

  // Define the scalable target
  RegisterScalableTargetRequest rstRequest = new RegisterScalableTargetRequest()
    .withServiceNamespace(ns)
    .withResourceId(resourceID)
    .withScalableDimension(tableWCUs)
    .withMinCapacity(5)
    .withMaxCapacity(10)
    .withRoleARN("SERVICE_ROLE_ARN_GOES_HERE");

  try {
   aaClient.registerScalableTarget(rstRequest);
  } catch (Exception e) {
   System.err.println("Unable to register scalable target: ");
   System.err.println(e.getMessage());
  }

  // Verify that the target was created
  DescribeScalableTargetsRequest dscRequest = new DescribeScalableTargetsRequest()
```

```
    .withServiceNamespace(ns)
    .withScalableDimension(tableWCUs)
    .withResourceIds(resourceID);
 try {
  DescribeScalableTargetsResult dsaResult =
aaClient.describeScalableTargets(dscRequest);
  System.out.println("DescribeScalableTargets result: ");
  System.out.println(dsaResult);
  System.out.println();
 } catch (Exception e) {
  System.err.println("Unable to describe scalable target: ");
  System.err.println(e.getMessage());
 }

 System.out.println();

 // Configure a scaling policy
 TargetTrackingScalingPolicyConfiguration targetTrackingScalingPolicyConfiguration
= new TargetTrackingScalingPolicyConfiguration()
    .withPredefinedMetricSpecification(
      new PredefinedMetricSpecification()
        .withPredefinedMetricType(MetricType.DynamoDBWriteCapacityUtilization))
    .withTargetValue(50.0)
    .withScaleInCooldown(60)
    .withScaleOutCooldown(60);

 // Create the scaling policy, based on your configuration
 PutScalingPolicyRequest pspRequest = new PutScalingPolicyRequest()
    .withServiceNamespace(ns)
    .withScalableDimension(tableWCUs)
    .withResourceId(resourceID)
    .withPolicyName("MyScalingPolicy")
    .withPolicyType(PolicyType.TargetTrackingScaling)

.withTargetTrackingScalingPolicyConfiguration(targetTrackingScalingPolicyConfiguration);

 try {
  aaClient.putScalingPolicy(pspRequest);
 } catch (Exception e) {
  System.err.println("Unable to put scaling policy: ");
  System.err.println(e.getMessage());
 }

 // Verify that the scaling policy was created
```

```
  DescribeScalingPoliciesRequest dspRequest = new DescribeScalingPoliciesRequest()
    .withServiceNamespace(ns)
    .withScalableDimension(tableWCUs)
    .withResourceId(resourceID);

  try {
   DescribeScalingPoliciesResult dspResult =
 aaClient.describeScalingPolicies(dspRequest);
   System.out.println("DescribeScalingPolicies result: ");
   System.out.println(dspResult);
  } catch (Exception e) {
   e.printStackTrace();
   System.err.println("Unable to describe scaling policy: ");
   System.err.println(e.getMessage());
  }

 }

}
```

Disabling Application Auto Scaling for a table

The following program reverses the previous process. It removes the auto scaling policy and then deregisters the scalable target.

Java v2

```
import software.amazon.awssdk.regions.Region;
import
 software.amazon.awssdk.services.applicationautoscaling.ApplicationAutoScalingClient;
import
 software.amazon.awssdk.services.applicationautoscaling.model.ApplicationAutoScalingExceptio
import
 software.amazon.awssdk.services.applicationautoscaling.model.DeleteScalingPolicyRequest;
import
 software.amazon.awssdk.services.applicationautoscaling.model.DeregisterScalableTargetReques
import
 software.amazon.awssdk.services.applicationautoscaling.model.DescribeScalableTargetsRequest
import
 software.amazon.awssdk.services.applicationautoscaling.model.DescribeScalableTargetsRespons
```

```
import
 software.amazon.awssdk.services.applicationautoscaling.model.DescribeScalingPoliciesRequest
import
 software.amazon.awssdk.services.applicationautoscaling.model.DescribeScalingPoliciesRespons
import
 software.amazon.awssdk.services.applicationautoscaling.model.ScalableDimension;
import
 software.amazon.awssdk.services.applicationautoscaling.model.ServiceNamespace;

/**
 * Before running this Java V2 code example, set up your development environment,
 including your credentials.
 *
 * For more information, see the following documentation topic:
 *
 * https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-started.html
 */

public class DisableDynamoDBAutoscaling {
    public static void main(String[] args) {
        final String usage = """

            Usage:
               <tableId> <policyName>\s

            Where:
               tableId - The table Id value (for example, table/Music).\s
               policyName - The name of the policy (for example, $Music5-scaling-
policy).

            """;
        if (args.length != 2) {
            System.out.println(usage);
            System.exit(1);
        }

        ApplicationAutoScalingClient appAutoScalingClient =
 ApplicationAutoScalingClient.builder()
            .region(Region.US_EAST_1)
            .build();

        ServiceNamespace ns = ServiceNamespace.DYNAMODB;
        ScalableDimension tableWCUs =
 ScalableDimension.DYNAMODB_TABLE_WRITE_CAPACITY_UNITS;
```

```
        String tableId = args[0];
        String policyName = args[1];

        deletePolicy(appAutoScalingClient, policyName, tableWCUs, ns, tableId);
        verifyScalingPolicies(appAutoScalingClient, tableId, ns, tableWCUs);
        deregisterScalableTarget(appAutoScalingClient, tableId, ns, tableWCUs);
        verifyTarget(appAutoScalingClient, tableId, ns, tableWCUs);
    }

    public static void deletePolicy(ApplicationAutoScalingClient
appAutoScalingClient, String policyName, ScalableDimension tableWCUs,
ServiceNamespace ns, String tableId) {
        try {
            DeleteScalingPolicyRequest delSPRequest =
DeleteScalingPolicyRequest.builder()
                .policyName(policyName)
                .scalableDimension(tableWCUs)
                .serviceNamespace(ns)
                .resourceId(tableId)
                .build();

            appAutoScalingClient.deleteScalingPolicy(delSPRequest);
            System.out.println(policyName +" was deleted successfully.");

        } catch (ApplicationAutoScalingException e) {
            System.err.println(e.awsErrorDetails().errorMessage());
        }
    }

    // Verify that the scaling policy was deleted
    public static void verifyScalingPolicies(ApplicationAutoScalingClient
appAutoScalingClient, String tableId, ServiceNamespace ns, ScalableDimension
tableWCUs) {
        DescribeScalingPoliciesRequest dscRequest =
DescribeScalingPoliciesRequest.builder()
            .scalableDimension(tableWCUs)
            .serviceNamespace(ns)
            .resourceId(tableId)
            .build();

        DescribeScalingPoliciesResponse response =
appAutoScalingClient.describeScalingPolicies(dscRequest);
        System.out.println("DescribeScalableTargets result: ");
        System.out.println(response);
```

```
    }

    public static void deregisterScalableTarget(ApplicationAutoScalingClient
 appAutoScalingClient, String tableId, ServiceNamespace ns, ScalableDimension
 tableWCUs) {
        try {
            DeregisterScalableTargetRequest targetRequest =
 DeregisterScalableTargetRequest.builder()
                .scalableDimension(tableWCUs)
                .serviceNamespace(ns)
                .resourceId(tableId)
                .build();

            appAutoScalingClient.deregisterScalableTarget(targetRequest);
            System.out.println("The scalable target was deregistered.");

        } catch (ApplicationAutoScalingException e) {
            System.err.println(e.awsErrorDetails().errorMessage());
        }
    }

    public static void verifyTarget(ApplicationAutoScalingClient
 appAutoScalingClient, String tableId, ServiceNamespace ns, ScalableDimension
 tableWCUs) {
        DescribeScalableTargetsRequest dscRequest =
 DescribeScalableTargetsRequest.builder()
            .scalableDimension(tableWCUs)
            .serviceNamespace(ns)
            .resourceIds(tableId)
            .build();

        DescribeScalableTargetsResponse response =
 appAutoScalingClient.describeScalableTargets(dscRequest);
        System.out.println("DescribeScalableTargets result: ");
        System.out.println(response);
    }
}
```

Java v1

```
package com.amazonaws.codesamples.autoscaling;
```

```
import
 com.amazonaws.services.applicationautoscaling.AWSApplicationAutoScalingClient;
import
 com.amazonaws.services.applicationautoscaling.model.DeleteScalingPolicyRequest;
import
 com.amazonaws.services.applicationautoscaling.model.DeregisterScalableTargetRequest;
import
 com.amazonaws.services.applicationautoscaling.model.DescribeScalableTargetsRequest;
import
 com.amazonaws.services.applicationautoscaling.model.DescribeScalableTargetsResult;
import
 com.amazonaws.services.applicationautoscaling.model.DescribeScalingPoliciesRequest;
import
 com.amazonaws.services.applicationautoscaling.model.DescribeScalingPoliciesResult;
import com.amazonaws.services.applicationautoscaling.model.ScalableDimension;
import com.amazonaws.services.applicationautoscaling.model.ServiceNamespace;

public class DisableDynamoDBAutoscaling {

 static AWSApplicationAutoScalingClient aaClient = new
 AWSApplicationAutoScalingClient();

 public static void main(String args[]) {

  ServiceNamespace ns = ServiceNamespace.Dynamodb;
  ScalableDimension tableWCUs = ScalableDimension.DynamodbTableWriteCapacityUnits;
  String resourceID = "table/TestTable";

  // Delete the scaling policy
  DeleteScalingPolicyRequest delSPRequest = new DeleteScalingPolicyRequest()
    .withServiceNamespace(ns)
    .withScalableDimension(tableWCUs)
    .withResourceId(resourceID)
    .withPolicyName("MyScalingPolicy");

  try {
   aaClient.deleteScalingPolicy(delSPRequest);
  } catch (Exception e) {
   System.err.println("Unable to delete scaling policy: ");
   System.err.println(e.getMessage());
  }

  // Verify that the scaling policy was deleted
```

```
 DescribeScalingPoliciesRequest descSPRequest = new
DescribeScalingPoliciesRequest()
   .withServiceNamespace(ns)
   .withScalableDimension(tableWCUs)
   .withResourceId(resourceID);

 try {
  DescribeScalingPoliciesResult dspResult =
aaClient.describeScalingPolicies(descSPRequest);
  System.out.println("DescribeScalingPolicies result: ");
  System.out.println(dspResult);
 } catch (Exception e) {
  e.printStackTrace();
  System.err.println("Unable to describe scaling policy: ");
  System.err.println(e.getMessage());
 }

 System.out.println();

 // Remove the scalable target
 DeregisterScalableTargetRequest delSTRequest = new
DeregisterScalableTargetRequest()
   .withServiceNamespace(ns)
   .withScalableDimension(tableWCUs)
   .withResourceId(resourceID);

 try {
  aaClient.deregisterScalableTarget(delSTRequest);
 } catch (Exception e) {
  System.err.println("Unable to deregister scalable target: ");
  System.err.println(e.getMessage());
 }

 // Verify that the scalable target was removed
 DescribeScalableTargetsRequest dscRequest = new DescribeScalableTargetsRequest()
   .withServiceNamespace(ns)
   .withScalableDimension(tableWCUs)
   .withResourceIds(resourceID);

 try {
  DescribeScalableTargetsResult dsaResult =
aaClient.describeScalableTargets(dscRequest);
  System.out.println("DescribeScalableTargets result: ");
  System.out.println(dsaResult);
```

```
    System.out.println();
   } catch (Exception e) {
    System.err.println("Unable to describe scalable target: ");
    System.err.println(e.getMessage());
   }

  }

}
```

DynamoDB reserved capacity

For provisioned capacity tables that use the Standard table class, DynamoDB offers the ability to purchase reserved capacity for your read and write capacity. A reserved capacity purchase is an agreement to pay for a minimum amount of provisioned throughput capacity, for the duration of the term of the agreement, in exchange for discounted pricing.

> **Note**
>
> You can't purchase reserved capacity for replicated write capacity units (rWCUs). Reserved capacity is applied only to the Region in which it was purchased. Reserved capacity is also not available for tables using the DynamoDB Standard-IA table class or on-demand capacity mode.

Reserved capacity is purchased in allocations of 100 WCUs or 100 RCUs. The smallest reserved capacity offering is 100 capacity units (reads or writes). DynamoDB reserved capacity is offered as either a one-year commitment or in select Regions as a three-year commitment. You can save up to 54% off standard rates for a one-year term and 77% off standard rates for a three-year term. For more information about how and when you should purchase, see Amazon DynamoDB Reserved Capacity.

> **Note**
>
> You can purchase up to a combined 1,000,000 reserved capacity units for write capacity units (WCUs) and read capacity units (RCUs) using the AWS Management Console. If you want to purchase more than 1,000,000 provisioned capacity units in a single purchase, or have active reserved capacity and want to purchase additional reserved capacity that

would result in more than 1,000,000 active provisioned capacity units, follow the process mentioned in "How to purchase reserved capacity" section in Amazon DynamoDB Reserved Capacity.

When you purchase DynamoDB reserved capacity, you pay a one-time partial upfront payment and receive a discounted hourly rate for the committed provisioned usage. You pay for the entire committed provisioned usage, regardless of actual usage, so your cost savings are closely tied to use. Any capacity that you provision in excess of the purchased reserved capacity is billed at standard provisioned capacity rates. By reserving your read and write capacity units ahead of time, you realize significant cost savings on your provisioned capacity costs.

You can't sell, cancel, or transfer reserved capacity to another Region or account.

Understanding DynamoDB warm throughput

Warm throughput refers to the number of read and write operations your DynamoDB table can instantaneously support. These values are available by default for all tables and global secondary indexes (GSI) and represent how much they have scaled based on historical usage. If you are using on-demand mode, or if you update your provisioned throughput to these values, your application will be able to issue requests up to those values instantly.

DynamoDB will automatically adjust warm throughput values as your usage increases. You can also increase these values proactively when needed, which is especially useful for upcoming peak events like product launches or sales. For planned peak events, where request rates to your DynamoDB table might increase by 10x, 100x, or more, you can now assess whether the current warm throughput is sufficient to handle the expected traffic. If it’s not, you can increase the warm throughput value without changing your throughput settings or billing mode. This process is referred to as *pre-warming* a table, allowing you to set a baseline that your tables can instantly support. This ensures your applications can handle higher request rates from the moment they occur. Once increased, warm throughput values can't be decreased.

You can increase the warm throughput value for read operations, write operations, or both. You can increase this value for new and existing single-Region tables, global tables, and GSIs. For global tables, this feature is available for version 2019.11.21 (Current), and the warm throughput settings you set will automatically apply to all replica tables in the global table. There is no limit to the number of DynamoDB tables you can pre-warm at any time. The time to complete pre-warming

depends on the values you set and the size of the table or index. You can submit simultaneous pre-warm requests and these requests will not interfere with any table operations. You can pre-warm your table up to the table or index quota limit for your account in that Region. Use the Service Quotas console to check your current limits and increase them if needed.

Warm throughput values are available by default for all tables and secondary indexes at no cost. However, if you proactively increase these default warm throughput values to pre-warm the tables, you will be charged for those requests. For more information, see Amazon DynamoDB pricing.

For more information about warm throughput, see the topics below:

Topics

- Check your DynamoDB table's current warm throughput
- Increase your existing DynamoDB table's warm throughput
- Create a new DynamoDB table with higher warm throughput
- Understanding DynamoDB warm throughput in different scenarios

Check your DynamoDB table's current warm throughput

Use the following AWS CLI and AWS Console instructions to check your table or index's current warm throughput value.

AWS Management Console

To check your DynamoDB table's warm throughput using the DynamoDB console:

1. Sign in to the AWS Management Console and open the DynamoDB console at https://console.aws.amazon.com/dynamodb/.
2. In the left navigation pane, choose Tables.
3. On the **Tables** page, choose your desired table.
4. Select **Additional settings** to view your current warm throughput value. This value is shown as read units per second and write units per second.

AWS CLI

The following AWS CLI example shows you how to check your DynamoDB table's warm throughput.

1. Run the `describe-table` operation on your DynamoDB table.

```
aws dynamodb describe-table --table-name GameScores
```

2. You'll receive a response similar to the one below. Your `WarmThroughput` settings will be displayed as `ReadUnitsPerSecond` and `WriteUnitsPerSecond`. The `Status` will be `UPDATING` when the warm throughput value is being updated, and `ACTIVE` when the new warm throughput value is set.

```
{
    "Table": {
        "AttributeDefinitions": [
            {
                "AttributeName": "GameTitle",
                "AttributeType": "S"
            },
            {
                "AttributeName": "TopScore",
                "AttributeType": "N"
            },
            {
                "AttributeName": "UserId",
                "AttributeType": "S"
            }
        ],
        "TableName": "GameScores",
        "KeySchema": [
            {
                "AttributeName": "UserId",
                "KeyType": "HASH"
            },
            {
                "AttributeName": "GameTitle",
                "KeyType": "RANGE"
            }
        ],
        "TableStatus": "ACTIVE",
        "CreationDateTime": 1726128388.729,
        "ProvisionedThroughput": {
            "NumberOfDecreasesToday": 0,
            "ReadCapacityUnits": 0,
            "WriteCapacityUnits": 0
        },
```

```
        "TableSizeBytes": 0,
        "ItemCount": 0,
        "TableArn": "arn:aws:dynamodb:us-east-1:XXXXXXXXXXXX:table/GameScores",
        "TableId": "XXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX",
        "BillingModeSummary": {
            "BillingMode": "PAY_PER_REQUEST",
            "LastUpdateToPayPerRequestDateTime": 1726128388.729
        },
        "GlobalSecondaryIndexes": [
            {
                "IndexName": "GameTitleIndex",
                "KeySchema": [
                    {
                        "AttributeName": "GameTitle",
                        "KeyType": "HASH"
                    },
                    {
                        "AttributeName": "TopScore",
                        "KeyType": "RANGE"
                    }
                ],
                "Projection": {
                    "ProjectionType": "INCLUDE",
                    "NonKeyAttributes": [
                        "UserId"
                    ]
                },
                "IndexStatus": "ACTIVE",
                "ProvisionedThroughput": {
                    "NumberOfDecreasesToday": 0,
                    "ReadCapacityUnits": 0,
                    "WriteCapacityUnits": 0
                },
                "IndexSizeBytes": 0,
                "ItemCount": 0,
                "IndexArn": "arn:aws:dynamodb:us-east-1:XXXXXXXXXXXX:table/
GameScores/index/GameTitleIndex",
                "WarmThroughput": {
                    "ReadUnitsPerSecond": 12000,
                    "WriteUnitsPerSecond": 4000,
                    "Status": "ACTIVE"
                }
            }
        ],
```

```
        "DeletionProtectionEnabled": false,
        "WarmThroughput": {
            "ReadUnitsPerSecond": 12000,
            "WriteUnitsPerSecond": 4000,
            "Status": "ACTIVE"
        }
    }
}
```

Increase your existing DynamoDB table's warm throughput

Once you've checked your DynamoDB table's current warm throughput value, you can update it with the following steps:

AWS Management Console

To check your DynamoDB table's warm throughput value using the DynamoDB console:

1. Sign in to the AWS Management Console and open the DynamoDB console at https://console.aws.amazon.com/dynamodb/.
2. In the left navigation pane, choose Tables.
3. On the **Tables** page, choose your desired table.
4. In the **Warm throughput** field, select **Edit**.
5. On the **Edit warm throughput** page, choose **Increase warm throughput**.
6. Adjust the **read units per second** and **write units pers second**. These two settings define the throughput your table can instantly handle.
7. Select **Save**.
8. Your **read units per second** and **write units per second** will be updated in the **Warm throughput** field when the request finishes processing.

> **Note**
>
> Updating your warm throughput value is an asynchronous task. The `Status` will change from `UPDATING` to `ACTIVE` when the update is complete.

AWS CLI

The following AWS CLI example shows you how to update your DynamoDB table's warm throughput value.

1. Run the `update-table` operation on your DynamoDB table.

```
aws dynamodb update-table \
    --table-name GameScores \
    --warm-throughput ReadUnitsPerSecond=12345,WriteUnitsPerSecond=4567 \
    --global-secondary-index-updates \
        "[
            {
                \"Update\": {
                    \"IndexName\": \"GameTitleIndex\",
                    \"WarmThroughput\": {
                        \"ReadUnitsPerSecond\": 88,
                        \"WriteUnitsPerSecond\": 77
                    }
                }
            }
        ]" \
    --region us-east-1
```

2. You'll receive a response similar to the one below. Your `WarmThroughput` settings will be displayed as `ReadUnitsPerSecond` and `WriteUnitsPerSecond`. The `Status` will be `UPDATING` when the warm throughput value is being updated, and `ACTIVE` when the new warm throughput value is set.

```
{
    "TableDescription": {
        "AttributeDefinitions": [
            {
                "AttributeName": "GameTitle",
                "AttributeType": "S"
            },
            {
                "AttributeName": "TopScore",
                "AttributeType": "N"
            },
            {
                "AttributeName": "UserId",
                "AttributeType": "S"
```

```
            }
        ],
        "TableName": "GameScores",
        "KeySchema": [
            {
                "AttributeName": "UserId",
                "KeyType": "HASH"
            },
            {
                "AttributeName": "GameTitle",
                "KeyType": "RANGE"
            }
        ],
        "TableStatus": "ACTIVE",
        "CreationDateTime": 1730242189.965,
        "ProvisionedThroughput": {
            "NumberOfDecreasesToday": 0,
            "ReadCapacityUnits": 20,
            "WriteCapacityUnits": 10
        },
        "TableSizeBytes": 0,
        "ItemCount": 0,
        "TableArn": "arn:aws:dynamodb:us-east-1:XXXXXXXXXXXX:table/GameScores",
        "TableId": "XXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX",
        "GlobalSecondaryIndexes": [
            {
                "IndexName": "GameTitleIndex",
                "KeySchema": [
                    {
                        "AttributeName": "GameTitle",
                        "KeyType": "HASH"
                    },
                    {
                        "AttributeName": "TopScore",
                        "KeyType": "RANGE"
                    }
                ],
                "Projection": {
                    "ProjectionType": "INCLUDE",
                    "NonKeyAttributes": [
                        "UserId"
                    ]
                },
                "IndexStatus": "ACTIVE",
```

```
                "ProvisionedThroughput": {
                    "NumberOfDecreasesToday": 0,
                    "ReadCapacityUnits": 50,
                    "WriteCapacityUnits": 25
                },
                "IndexSizeBytes": 0,
                "ItemCount": 0,
                "IndexArn": "arn:aws:dynamodb:us-east-1:XXXXXXXXXXXX:table/
GameScores/index/GameTitleIndex",
                "WarmThroughput": {
                    "ReadUnitsPerSecond": 50,
                    "WriteUnitsPerSecond": 25,
                    "Status": "UPDATING"
                }
            }
        ],
        "DeletionProtectionEnabled": false,
        "WarmThroughput": {
            "ReadUnitsPerSecond": 12300,
            "WriteUnitsPerSecond": 4500,
            "Status": "UPDATING"
        }
    }
}
```

AWS SDK

The following SDK examples shows you how to update your DynamoDB table's warm throughput value.

Java

```
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.GlobalSecondaryIndexUpdate;
import
 software.amazon.awssdk.services.dynamodb.model.UpdateGlobalSecondaryIndexAction;
import software.amazon.awssdk.services.dynamodb.model.UpdateTableRequest;
import software.amazon.awssdk.services.dynamodb.model.WarmThroughput;

...
public static WarmThroughput buildWarmThroughput(final Long readUnitsPerSecond,
```

```
                                                    final Long writeUnitsPerSecond) {
    return WarmThroughput.builder()
            .readUnitsPerSecond(readUnitsPerSecond)
            .writeUnitsPerSecond(writeUnitsPerSecond)
            .build();
}

public static void updateDynamoDBTable(DynamoDbClient ddb,
                                       String tableName,
                                       Long tableReadUnitsPerSecond,
                                       Long tableWriteUnitsPerSecond,
                                       String globalSecondaryIndexName,
                                       Long globalSecondaryIndexReadUnitsPerSecond,
                                       Long globalSecondaryIndexWriteUnitsPerSecond)
 {

    final WarmThroughput tableWarmThroughput =
 buildWarmThroughput(tableReadUnitsPerSecond, tableWriteUnitsPerSecond);
    final WarmThroughput gsiWarmThroughput =
 buildWarmThroughput(globalSecondaryIndexReadUnitsPerSecond,
 globalSecondaryIndexWriteUnitsPerSecond);

    final GlobalSecondaryIndexUpdate globalSecondaryIndexUpdate =
 GlobalSecondaryIndexUpdate.builder()
            .update(UpdateGlobalSecondaryIndexAction.builder()
                    .indexName(globalSecondaryIndexName)
                    .warmThroughput(gsiWarmThroughput)
                    .build()
            ).build();

    final UpdateTableRequest request = UpdateTableRequest.builder()
            .tableName(tableName)
            .globalSecondaryIndexUpdates(globalSecondaryIndexUpdate)
            .warmThroughput(tableWarmThroughput)
            .build();

    try {
        ddb.updateTable(request);
    } catch (DynamoDbException e) {
        System.err.println(e.getMessage());
        System.exit(1);
    }

    System.out.println("Done!");
```

```
}
```

Python

```
from boto3 import resource
from botocore.exceptions import ClientError

def update_dynamodb_table_warm_throughput(table_name, table_read_units,
 table_write_units, gsi_name, gsi_read_units, gsi_write_units, region_name="us-
east-1"):
    """
    Updates the warm throughput of a DynamoDB table and a global secondary index.

    :param table_name: The name of the table to update.
    :param table_read_units: The new read units per second for the table's warm
 throughput.
    :param table_write_units: The new write units per second for the table's warm
 throughput.
    :param gsi_name: The name of the global secondary index to update.
    :param gsi_read_units: The new read units per second for the GSI's warm
 throughput.
    :param gsi_write_units: The new write units per second for the GSI's warm
 throughput.
    :param region_name: The AWS Region name to target. defaults to us-east-1
    """
    try:
        ddb = resource('dynamodb', region_name)

        # Update the table's warm throughput
        table_warm_throughput = {
            "ReadUnitsPerSecond": table_read_units,
            "WriteUnitsPerSecond": table_write_units
        }

        # Update the global secondary index's warm throughput
        gsi_warm_throughput = {
            "ReadUnitsPerSecond": gsi_read_units,
            "WriteUnitsPerSecond": gsi_write_units
        }

        # Construct the global secondary index update
        global_secondary_index_update = [
            {
```

```
                "Update": {
                    "IndexName": gsi_name,
                    "WarmThroughput": gsi_warm_throughput
                }
            }
        ]

        # Construct the update table request
        update_table_request = {
            "TableName": table_name,
            "GlobalSecondaryIndexUpdates": global_secondary_index_update,
            "WarmThroughput": table_warm_throughput
        }

        # Update the table
        ddb.update_table(**update_table_request)
        print("Table updated successfully!")
    except ClientError as e:
        print(f"Error updating table: {e}")
        raise e
```

Javascript

```
import { DynamoDBClient, UpdateTableCommand } from "@aws-sdk/client-dynamodb";

async function updateDynamoDBTableWarmThroughput(
  tableName,
  tableReadUnits,
  tableWriteUnits,
  gsiName,
  gsiReadUnits,
  gsiWriteUnits,
  region = "us-east-1"
) {
  try {
    const ddbClient = new DynamoDBClient({ region: region });

    // Construct the update table request
    const updateTableRequest = {
      TableName: tableName,
      GlobalSecondaryIndexUpdates: [
        {
            Update: {
```

```
                IndexName: gsiName,
                WarmThroughput: {
                    ReadUnitsPerSecond: gsiReadUnits,
                    WriteUnitsPerSecond: gsiWriteUnits,
                },
            },
        },
      ],
      WarmThroughput: {
          ReadUnitsPerSecond: tableReadUnits,
          WriteUnitsPerSecond: tableWriteUnits,
      },
    };

    const command = new UpdateTableCommand(updateTableRequest);
    const response = await ddbClient.send(command);
    console.log(`Table updated successfully! Response: ${response}`);
  } catch (error) {
    console.error(`Error updating table: ${error}`);
    throw error;
  }
}
```

Create a new DynamoDB table with higher warm throughput

You can adjust the warm throughput values when you create your DynamoDB table by following the steps below. These steps also apply when creating a global table or secondary index.

AWS Management Console

To create a DynamoDB table and adjust the warm throughput values through the console:

1. Sign in to the AWS Management Console and open the DynamoDB console at https://console.aws.amazon.com/dynamodb/.
2. Select **Create table**.
3. Choose a **Table name**, **Partition key**, and **Sort key (optional)**.
4. For **Table settings**, choose **Customize settings**.
5. In the **Warm throughput** field, choose **Increase warm throughput.**
6. Adjust the **read units per second** and **write units pers second**. These two settings define the maximum throughput your table can instantly handle.

7. Continue adding any remaining table details and then select **Create table**.

AWS CLI

The following AWS CLI example shows you how to create a DynamoDB table with customized warm throughput values.

1. Run the `create-table` operation to create the following DynamoDB table.

```
aws dynamodb create-table \
    --table-name GameScores \
    --attribute-definitions AttributeName=UserId,AttributeType=S \
                            AttributeName=GameTitle,AttributeType=S \
                            AttributeName=TopScore,AttributeType=N  \
    --key-schema AttributeName=UserId,KeyType=HASH \
                 AttributeName=GameTitle,KeyType=RANGE \
    --provisioned-throughput ReadCapacityUnits=20,WriteCapacityUnits=10 \
    --global-secondary-indexes \
        "[
            {
                \"IndexName\": \"GameTitleIndex\",
                \"KeySchema\": [{\"AttributeName\":\"GameTitle\",\"KeyType\":\"HASH
\"},
                                {\"AttributeName\":\"TopScore\",\"KeyType\":\"RANGE
\"}],
                \"Projection\":{
                    \"ProjectionType\":\"INCLUDE\",
                    \"NonKeyAttributes\":[\"UserId\"]
                },
                \"ProvisionedThroughput\": {
                    \"ReadCapacityUnits\": 50,
                    \"WriteCapacityUnits\": 25
                },\"WarmThroughput\": {
                    \"ReadUnitsPerSecond\": 1987,
                    \"WriteUnitsPerSecond\": 543
                }
            }
        ]" \
    --warm-throughput ReadUnitsPerSecond=12345,WriteUnitsPerSecond=4567 \
    --region us-east-1
```

2. You'll receive a response similar to the one below. Your `WarmThroughput` settings will be displayed as `ReadUnitsPerSecond` and `WriteUnitsPerSecond`. The `Status` will be `UPDATING` when the warm throughput value is being updated, and `ACTIVE` when the new warm throughput value is set.


```
{
    "TableDescription": {
        "AttributeDefinitions": [
            {
                "AttributeName": "GameTitle",
                "AttributeType": "S"
            },
            {
                "AttributeName": "TopScore",
                "AttributeType": "N"
            },
            {
                "AttributeName": "UserId",
                "AttributeType": "S"
            }
        ],
        "TableName": "GameScores",
        "KeySchema": [
            {
                "AttributeName": "UserId",
                "KeyType": "HASH"
            },
            {
                "AttributeName": "GameTitle",
                "KeyType": "RANGE"
            }
        ],
        "TableStatus": "CREATING",
        "CreationDateTime": 1730241788.779,
        "ProvisionedThroughput": {
            "NumberOfDecreasesToday": 0,
            "ReadCapacityUnits": 20,
            "WriteCapacityUnits": 10
        },
        "TableSizeBytes": 0,
        "ItemCount": 0,
        "TableArn": "arn:aws:dynamodb:us-east-1:XXXXXXXXXXXX:table/GameScores",
        "TableId": "XXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX",
```

```
        "GlobalSecondaryIndexes": [
            {
                "IndexName": "GameTitleIndex",
                "KeySchema": [
                    {
                        "AttributeName": "GameTitle",
                        "KeyType": "HASH"
                    },
                    {
                        "AttributeName": "TopScore",
                        "KeyType": "RANGE"
                    }
                ],
                "Projection": {
                    "ProjectionType": "INCLUDE",
                    "NonKeyAttributes": [
                        "UserId"
                    ]
                },
                "IndexStatus": "CREATING",
                "ProvisionedThroughput": {
                    "NumberOfDecreasesToday": 0,
                    "ReadCapacityUnits": 50,
                    "WriteCapacityUnits": 25
                },
                "IndexSizeBytes": 0,
                "ItemCount": 0,
                "IndexArn": "arn:aws:dynamodb:us-east-1:XXXXXXXXXXXX:table/
GameScores/index/GameTitleIndex",
                "WarmThroughput": {
                    "ReadUnitsPerSecond": 1987,
                    "WriteUnitsPerSecond": 543,
                    "Status": "UPDATING"
                }
            }
        ],
        "DeletionProtectionEnabled": false,
        "WarmThroughput": {
            "ReadUnitsPerSecond": 12345,
            "WriteUnitsPerSecond": 4567,
            "Status": "UPDATING"
        }
    }
```

```
}
```

AWS SDK

The following SDK examples shows you how to create a DynamoDB table with customized warm throughput values.

Java

```
import software.amazon.awscdk.services.dynamodb.ProjectionType;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.CreateTableResponse;
import software.amazon.awssdk.services.dynamodb.model.CreateTableRequest;
import software.amazon.awssdk.services.dynamodb.model.KeySchemaElement;
import software.amazon.awssdk.services.dynamodb.model.KeyType;
import software.amazon.awssdk.services.dynamodb.model.ProvisionedThroughput;
import software.amazon.awssdk.services.dynamodb.model.Projection;
import software.amazon.awssdk.services.dynamodb.model.GlobalSecondaryIndex;
import software.amazon.awssdk.services.dynamodb.model.AttributeDefinition;
import software.amazon.awssdk.services.dynamodb.model.ScalarAttributeType;
import software.amazon.awssdk.services.dynamodb.model.WarmThroughput;
...

public static WarmThroughput buildWarmThroughput(final Long readUnitsPerSecond,
                                                 final Long writeUnitsPerSecond) {
    return WarmThroughput.builder()
            .readUnitsPerSecond(readUnitsPerSecond)
            .writeUnitsPerSecond(writeUnitsPerSecond)
            .build();
}
private static AttributeDefinition buildAttributeDefinition(final String
 attributeName,
                                                             final
 ScalarAttributeType scalarAttributeType) {
    return AttributeDefinition.builder()
            .attributeName(attributeName)
            .attributeType(scalarAttributeType)
            .build();
}
private static KeySchemaElement buildKeySchemaElement(final String attributeName,
                                                      final KeyType keyType) {
    return KeySchemaElement.builder()
```

```
            .attributeName(attributeName)
            .keyType(keyType)
            .build();
}
public static void createDynamoDBTable(DynamoDbClient ddb,
                                        String tableName,
                                        String partitionKey,
                                        String sortKey,
                                        String miscellaneousKeyAttribute,
                                        String nonKeyAttribute,
                                        Long tableReadCapacityUnits,
                                        Long tableWriteCapacityUnits,
                                        Long tableWarmReadUnitsPerSecond,
                                        Long tableWarmWriteUnitsPerSecond,
                                        String globalSecondaryIndexName,
                                        Long globalSecondaryIndexReadCapacityUnits,
                                        Long globalSecondaryIndexWriteCapacityUnits,
                                        Long
 globalSecondaryIndexWarmReadUnitsPerSecond,
                                        Long
 globalSecondaryIndexWarmWriteUnitsPerSecond) {

   // Define the table attributes
   final AttributeDefinition partitionKeyAttribute =
 buildAttributeDefinition(partitionKey, ScalarAttributeType.S);
   final AttributeDefinition sortKeyAttribute = buildAttributeDefinition(sortKey,
 ScalarAttributeType.S);
   final AttributeDefinition miscellaneousKeyAttributeDefinition =
 buildAttributeDefinition(miscellaneousKeyAttribute, ScalarAttributeType.N);
   final AttributeDefinition[] attributeDefinitions = {partitionKeyAttribute,
 sortKeyAttribute, miscellaneousKeyAttributeDefinition};

   // Define the table key schema
   final KeySchemaElement partitionKeyElement = buildKeySchemaElement(partitionKey,
 KeyType.HASH);
   final KeySchemaElement sortKeyElement = buildKeySchemaElement(sortKey,
 KeyType.RANGE);
   final KeySchemaElement[] keySchema = {partitionKeyElement, sortKeyElement};

   // Define the provisioned throughput for the table
   final ProvisionedThroughput provisionedThroughput =
 ProvisionedThroughput.builder()
            .readCapacityUnits(tableReadCapacityUnits)
            .writeCapacityUnits(tableWriteCapacityUnits)
```

```
            .build();

   // Define the Global Secondary Index (GSI)
   final KeySchemaElement globalSecondaryIndexPartitionKeyElement =
buildKeySchemaElement(sortKey, KeyType.HASH);
   final KeySchemaElement globalSecondaryIndexSortKeyElement =
buildKeySchemaElement(miscellaneousKeyAttribute, KeyType.RANGE);
   final KeySchemaElement[] gsiKeySchema =
{globalSecondaryIndexPartitionKeyElement, globalSecondaryIndexSortKeyElement};

   final Projection gsiProjection = Projection.builder()
           .projectionType(String.valueOf(ProjectionType.INCLUDE))
           .nonKeyAttributes(nonKeyAttribute)
           .build();
   final ProvisionedThroughput gsiProvisionedThroughput =
ProvisionedThroughput.builder()
           .readCapacityUnits(globalSecondaryIndexReadCapacityUnits)
           .writeCapacityUnits(globalSecondaryIndexWriteCapacityUnits)
           .build();
   // Define the warm throughput for the Global Secondary Index (GSI)
   final WarmThroughput gsiWarmThroughput =
buildWarmThroughput(globalSecondaryIndexWarmReadUnitsPerSecond,
globalSecondaryIndexWarmWriteUnitsPerSecond);
   final GlobalSecondaryIndex globalSecondaryIndex = GlobalSecondaryIndex.builder()
           .indexName(globalSecondaryIndexName)
           .keySchema(gsiKeySchema)
           .projection(gsiProjection)
           .provisionedThroughput(gsiProvisionedThroughput)
           .warmThroughput(gsiWarmThroughput)
           .build();

   // Define the warm throughput for the table
   final WarmThroughput tableWarmThroughput =
buildWarmThroughput(tableWarmReadUnitsPerSecond, tableWarmWriteUnitsPerSecond);

   final CreateTableRequest request = CreateTableRequest.builder()
           .tableName(tableName)
           .attributeDefinitions(attributeDefinitions)
           .keySchema(keySchema)
           .provisionedThroughput(provisionedThroughput)
           .globalSecondaryIndexes(globalSecondaryIndex)
           .warmThroughput(tableWarmThroughput)
           .build();
```

```
    CreateTableResponse response = ddb.createTable(request);
    System.out.println(response);
}
```

Python

```
from boto3 import resource
from botocore.exceptions import ClientError

def create_dynamodb_table_warm_throughput(table_name, partition_key,
 sort_key, misc_key_attr, non_key_attr, table_provisioned_read_units,
 table_provisioned_write_units, table_warm_reads, table_warm_writes, gsi_name,
 gsi_provisioned_read_units, gsi_provisioned_write_units, gsi_warm_reads,
 gsi_warm_writes, region_name="us-east-1"):
    """
    Creates a DynamoDB table with a warm throughput setting configured.

    :param table_name: The name of the table to be created.
    :param partition_key: The partition key for the table being created.
    :param sort_key: The sort key for the table being created.
    :param misc_key_attr: A miscellaneous key attribute for the table being created.
    :param non_key_attr: A non-key attribute for the table being created.
    :param table_provisioned_read_units: The newly created table's provisioned read
 capacity units.
    :param table_provisioned_write_units: The newly created table's provisioned
 write capacity units.
    :param table_warm_reads: The read units per second setting for the table's warm
 throughput.
    :param table_warm_writes: The write units per second setting for the table's
 warm throughput.
    :param gsi_name: The name of the Global Secondary Index (GSI) to be created on
 the table.
    :param gsi_provisioned_read_units: The configured Global Secondary Index (GSI)
 provisioned read capacity units.
    :param gsi_provisioned_write_units: The configured Global Secondary Index (GSI)
 provisioned write capacity units.
    :param gsi_warm_reads: The read units per second setting for the Global
 Secondary Index (GSI)'s warm throughput.
    :param gsi_warm_writes: The write units per second setting for the Global
 Secondary Index (GSI)'s warm throughput.
    :param region_name: The AWS Region name to target. defaults to us-east-1
    """
    try:
```

```
        ddb = resource('dynamodb', region_name)

        # Define the table attributes
        attribute_definitions = [
            { "AttributeName": partition_key, "AttributeType": "S" },
            { "AttributeName": sort_key, "AttributeType": "S" },
            { "AttributeName": misc_key_attr, "AttributeType": "N" }
        ]

        # Define the table key schema
        key_schema = [
            { "AttributeName": partition_key, "KeyType": "HASH" },
            { "AttributeName": sort_key, "KeyType": "RANGE" }
        ]

        # Define the provisioned throughput for the table
        provisioned_throughput = {
            "ReadCapacityUnits": table_provisioned_read_units,
            "WriteCapacityUnits": table_provisioned_write_units
        }

        # Define the global secondary index
        gsi_key_schema = [
            { "AttributeName": sort_key, "KeyType": "HASH" },
            { "AttributeName": misc_key_attr, "KeyType": "RANGE" }
        ]
        gsi_projection = {
            "ProjectionType": "INCLUDE",
            "NonKeyAttributes": [non_key_attr]
        }
        gsi_provisioned_throughput = {
            "ReadCapacityUnits": gsi_provisioned_read_units,
            "WriteCapacityUnits": gsi_provisioned_write_units
        }
        gsi_warm_throughput = {
            "ReadUnitsPerSecond": gsi_warm_reads,
            "WriteUnitsPerSecond": gsi_warm_writes
        }
        global_secondary_indexes = [
            {
                "IndexName": gsi_name,
                "KeySchema": gsi_key_schema,
                "Projection": gsi_projection,
                "ProvisionedThroughput": gsi_provisioned_throughput,
```

```
                "WarmThroughput": gsi_warm_throughput
            }
        ]

        # Define the warm throughput for the table
        warm_throughput = {
            "ReadUnitsPerSecond": table_warm_reads,
            "WriteUnitsPerSecond": table_warm_writes
        }

        # Create the DynamoDB client and create the table
        response = ddb.create_table(
            TableName=table_name,
            AttributeDefinitions=attribute_definitions,
            KeySchema=key_schema,
            ProvisionedThroughput=provisioned_throughput,
            GlobalSecondaryIndexes=global_secondary_indexes,
            WarmThroughput=warm_throughput
        )

        print(response)
    except ClientError as e:
        print(f"Error creating table: {e}")
        raise e
```

Javascript

```
import { DynamoDBClient, CreateTableCommand } from "@aws-sdk/client-dynamodb";

async function createDynamoDBTableWithWarmThroughput(
  tableName,
  partitionKey,
  sortKey,
  miscKeyAttr,
  nonKeyAttr,
  tableProvisionedReadUnits,
  tableProvisionedWriteUnits,
  tableWarmReads,
  tableWarmWrites,
  indexName,
  indexProvisionedReadUnits,
  indexProvisionedWriteUnits,
  indexWarmReads,
```

```
  indexWarmWrites,
  region = "us-east-1"
) {
  try {
    const ddbClient = new DynamoDBClient({ region: region });
    const command = new CreateTableCommand({
      TableName: tableName,
      AttributeDefinitions: [
          { AttributeName: partitionKey, AttributeType: "S" },
          { AttributeName: sortKey, AttributeType: "S" },
          { AttributeName: miscKeyAttr, AttributeType: "N" },
      ],
      KeySchema: [
          { AttributeName: partitionKey, KeyType: "HASH" },
          { AttributeName: sortKey, KeyType: "RANGE" },
      ],
      ProvisionedThroughput: {
          ReadCapacityUnits: tableProvisionedReadUnits,
          WriteCapacityUnits: tableProvisionedWriteUnits,
      },
      WarmThroughput: {
          ReadUnitsPerSecond: tableWarmReads,
          WriteUnitsPerSecond: tableWarmWrites,
      },
      GlobalSecondaryIndexes: [
          {
            IndexName: indexName,
            KeySchema: [
                { AttributeName: sortKey, KeyType: "HASH" },
                { AttributeName: miscKeyAttr, KeyType: "RANGE" },
            ],
            Projection: {
                ProjectionType: "INCLUDE",
                NonKeyAttributes: [nonKeyAttr],
            },
            ProvisionedThroughput: {
                ReadCapacityUnits: indexProvisionedReadUnits,
                WriteCapacityUnits: indexProvisionedWriteUnits,
            },
            WarmThroughput: {
                ReadUnitsPerSecond: indexWarmReads,
                WriteUnitsPerSecond: indexWarmWrites,
            },
          },
```

```
      ],
    });
    const response = await ddbClient.send(command);
    console.log(response);
  } catch (error) {
    console.error(`Error creating table: ${error}`);
    throw error;
  }
}
```

Understanding DynamoDB warm throughput in different scenarios

Here are some different scenarios you might encounter when working with DynamoDB warm throughput.

Topics

- Warm throughput and uneven access patterns
- Warm throughput for a provisioned table
- Warm throughput for an on-demand table
- Warm throughput for an on-demand table with maximum throughput configured

Warm throughput and uneven access patterns

A table might have a warm throughput of 30,000 read units per second and 10,000 write units per second, but you could still experience throttling on reads or writes before hitting those values. This is likely due to a hot partition. While DynamoDB can keep scaling to support virtually unlimited throughput, each individual partition is limited to 1,000 write units per second and 3,000 read units per second. If your application drives too much traffic to a small portion of the table's partitions, throttling can occur even before you reach the table's warm throughput values. We recommend following DynamoDB best practices to ensure seamless scalability and avoid hot partitions.

Warm throughput for a provisioned table

Consider a provisioned table that has a warm throughput of 30,000 read units per second and 10,000 write units per second but currently has a provisioned throughput of 4,000 RCU and 8,000 WCU. You can instantly scale the table's provisioned throughput up to 30,000 RCU or 10,000 WCU by updating your provisioned throughput settings. As you increase the provisioned throughput

beyond these values, the warm throughput will automatically adjust to the new higher values, because you have established a new peak throughput. For example, if you set the provisioned throughput to 50,000 RCU, the warm throughput will increase to 50,000 read units per second.

```
"ProvisionedThroughput":
    {
        "ReadCapacityUnits": 4000,
        "WriteCapacityUnits": 8000
    }
"WarmThroughput":
    {
        "ReadUnitsPerSecond": 30000,
        "WriteUnitsPerSecond": 10000
    }
```

Warm throughput for an on-demand table

A new on-demand table starts with a warm throughput of 12,000 read units per second and 4,000 write units per second. Your table can instantly accommodate sustained traffic up to these levels. When your requests exceed 12,000 read units per second or 4,000 write units per second, the warm throughput will automatically adjust to higher values.

```
"WarmThroughput":
    {
        "ReadUnitsPerSecond": 12000,
        "WriteUnitsPerSecond": 4000
    }
```

Warm throughput for an on-demand table with maximum throughput configured

Consider an on-demand table with a warm throughput of 30,000 read units per second but with a maximum throughput configured at 5,000 read request units (RRU). In this scenario, the table's throughput will be limited to the maximum of 5,000 RRU that you set. Any throughput requests exceeding this maximum will be throttled. However, you can modify the table-specific maximum throughput at any time based on your application's needs.

```
"OnDemandThroughput":
    {
        "MaxReadRequestUnits": 5000,
        "MaxWriteRequestUnits": 4000
    }
```

```
"WarmThroughput":
    {
        "ReadUnitsPerSecond": 30000,
        "WriteUnitsPerSecond": 10000
    }
```

DynamoDB burst and adaptive capacity

To minimize throttling because of throughput exceptions, DynamoDB uses *burst capacity* to handle usage spikes. DynamoDB uses *adaptive capacity* to help accommodate uneven access patterns.

Burst capacity

DynamoDB provides some flexibility for your throughput provisioning with *burst capacity*. Whenever you aren't fully using your available throughput, DynamoDB reserves a portion of that unused capacity for later *bursts* of throughput to handle usage spikes. With burst capacity, unexpected read or write requests can succeed where they otherwise would be throttled.

DynamoDB currently retains up to five minutes (300 seconds) of unused read and write capacity. During an occasional burst of read or write activity, these extra capacity units can be consumed quickly — even faster than the per-second provisioned throughput capacity that you've defined for your table.

DynamoDB can also consume burst capacity for background maintenance and other tasks without prior notice.

Note that these burst capacity details might change in the future.

Adaptive capacity

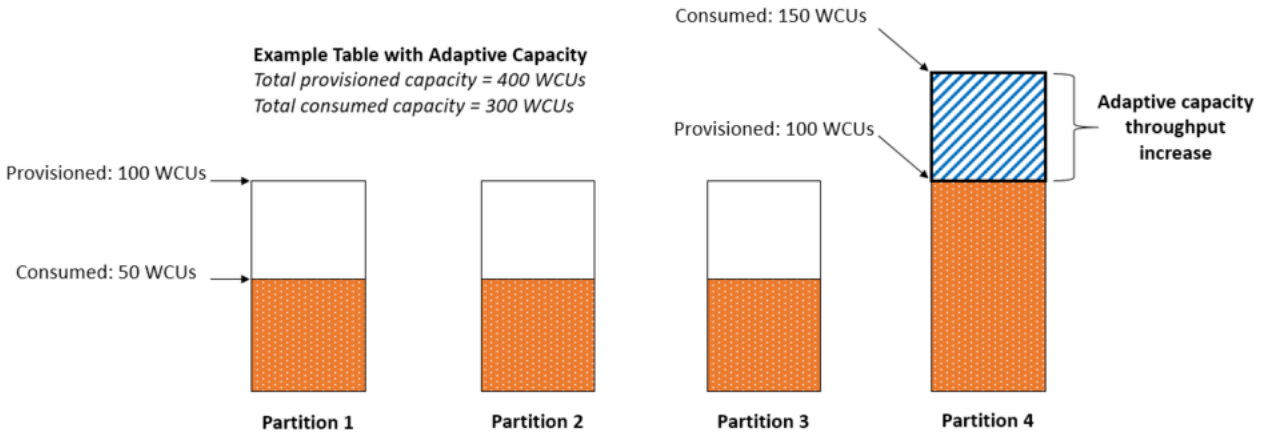
DynamoDB automatically distributes your data across partitions, which are stored on multiple servers in the AWS Cloud. It's not always possible to evenly distribute read and write activity all the time. When data access is imbalanced, a "hot" partition can receive a higher volume of read and write traffic compared to other partitions. Because read and write operations on a partition are managed independently, throttling will occur if a single partition receives more than 3000 read operation or more than 1000 write operations. Adaptive capacity works by automatically increasing throughput capacity for partitions that receive more traffic.

To better accommodate uneven access patterns, DynamoDB adaptive capacity enables your application to continue reading and writing to hot partitions without being throttled, provided

that traffic does not exceed your table's total provisioned capacity or the partition maximum capacity. Adaptive capacity works by automatically and instantly increasing throughput capacity for partitions that receive more traffic.

The following diagram illustrates how adaptive capacity works. The example table is provisioned with 400 WCUs evenly shared across four partitions, allowing each partition to sustain up to 100 WCUs per second. Partitions 1, 2, and 3 each receives write traffic of 50 WCU/sec. Partition 4 receives 150 WCU/sec. This hot partition can accept write traffic while it still has unused burst capacity, but eventually it throttles traffic that exceeds 100 WCU/sec.

DynamoDB adaptive capacity responds by increasing the capacity of partition 4 so that it can sustain the higher workload of 150 WCU/sec without being throttled.



Adaptive capacity is enabled automatically for every DynamoDB table, at no additional cost. You don't need to explicitly enable or disable it.

Isolate frequently accessed items

If your application drives disproportionately high traffic to one or more items, adaptive capacity rebalances your partitions such that frequently accessed items don't reside on the same partition. This isolation of frequently accessed items reduces the likelihood of request throttling due to your workload exceeding the throughput quota on a single partition. You can also break up an item collection into segments by sort key, as long as the item collection isn't traffic that is tracked by a monotonic increase or decrease of the sort key.

If your application drives consistently high traffic to a single item, adaptive capacity might rebalance your data so that a partition contains only that single, frequently accessed item. In this case, DynamoDB can deliver throughput up to the partition maximum of 3,000 RCUs and 1,000

WCUs to that single item's primary key. Adaptive capacity will not split item collections across multiple partitions of the table when there is a local secondary index on the table.

Considerations when switching capacity modes in DynamoDB

When you create a DynamoDB table, you must select either on-demand or provisioned capacity mode.

You can switch tables from provisioned capacity mode to on-demand mode up to four times in a 24-hour rolling window. You can switch tables from on-demand mode to provisioned capacity mode at any time.

Topics

- Switching from provisioned capacity mode to on-demand capacity mode
- Switching from on-demand capacity mode to provisioned capacity mode

Switching from provisioned capacity mode to on-demand capacity mode

In provisioned mode, you set read and write capacity based on your expected application needs. When you update a table from provisioned to on-demand mode, you don't need to specify how much read and write throughput you expect your application to perform. DynamoDB on-demand offers simple pay-per-request pricing for read and write requests so that you only pay for what you use, making it easy to balance costs and performance. You can optionally configure maximum read or write (or both) throughput for individual on-demand tables and associated global secondary indexes to help keep costs and usage bounded. For more information about setting maximum throughput for a specific table or index, see DynamoDB maximum throughput for on-demand tables.

When you switch from provisioned capacity mode to on-demand capacity mode, DynamoDB makes several changes to the structure of your table and partitions. This process can take several minutes. During the switching period, your table delivers throughput that is consistent with the previously provisioned write capacity unit and read capacity unit amounts.

Initial throughput for on-demand capacity mode

If you recently switched an existing table to on-demand capacity mode for the first time, the table has the following previous peak settings, even though the table has not served traffic previously using on-demand capacity mode.

Following are examples of possible scenarios:

- **Any provisioned table configured below 4000 WCU and 12,000 RCU, that has never been previously provisioned for more.** When you switch this table to on-demand for the first time, DynamoDB will ensure it is scaled out to instantly sustain at least 4,000 write units/sec and 12,000 read units/sec.
- **A provisioned table configured as 8,000 WCU and 24,000 RCU.** When you switch this table to on-demand, it will continue to be able to sustain at least 8,000 write units/sec and 24,000 read units/sec at any time.
- **A provisioned table configured with 8,000 WCU and 24,000 RCU, that consumed 6,000 write units/sec and 18,000 read units/sec for a sustained period.** When you switch this table to on-demand, it will continue to be able to sustain at least 8,000 write units/sec and 24,000 read units/sec. The previous traffic may further allow the table to sustain much higher levels of traffic without throttling.
- **A table previously provisioned with 10,000 WCU and 10,000 RCU, but currently provisioned with 10 RCU and 10 WCU.** When you switch this table to on-demand, it will be able to sustain at least 10,000 write units/sec and 10,000 read units/sec.

Auto scaling settings

When you update a table from provisioned to on-demand mode:

- If you're using the console, all of your auto scaling settings (if any) will be deleted.
- If you're using the AWS CLI or AWS SDK, all of your auto scaling settings will be preserved. These settings can be applied when you update your table to provisioned billing mode again.

Bulk editing capacity mode in the DynamoDB console

You can bulk edit multiple tables to switch from provisioned capacity mode to on-demand capacity mode using the DynamoDB console. To bulk edit capacity mode:

1. In the DynamoDB console, go to the **Tables** page.
2. Select the checkboxes for the tables you want to update to on-demand capacity mode.
3. Choose **Actions**, and then select **Update to on-demand capacity mode**.

This bulk operation allows you to efficiently switch multiple tables to on-demand capacity mode without having to update each table individually.

Switching from on-demand capacity mode to provisioned capacity mode

When switching from on-demand capacity mode back to provisioned capacity mode, your table delivers throughput consistent with the previous peak reached when the table was set to on-demand capacity mode.

Managing capacity

Consider the following when you update a table from on-demand to provisioned mode:

- If you're using the AWS CLI or AWS SDK, choose the right provisioned capacity settings of your table and global secondary indexes by using Amazon CloudWatch to look at your historical consumption (`ConsumedWriteCapacityUnits` and `ConsumedReadCapacityUnits` metrics) to determine the new throughput settings.

  > **Note**
  >
  > If you're switching a global table to provisioned mode, look at the maximum consumption across all your regional replicas for base tables and global secondary indexes when determining the new throughput settings.

- If you're switching from on-demand mode back to provisioned mode, make sure to set the initial provisioned units high enough to handle your table or index capacity during the transition.

Managing auto scaling

When you update a table from on-demand to provisioned mode:

- If you're using the console, we recommend enabling auto scaling with the following defaults:

  - Target utilization: 70%
  - Minimum provisioned capacity: 5 units
  - Maximum provisioned capacity: The Region maximum
- If you're using the AWS CLI or SDK, your previous auto scaling settings (if any) are preserved.

Programming with DynamoDB and the AWS SDKs

This section covers developer-related topics. If you want to run code examples instead, see Running the code examples in this Developer Guide.

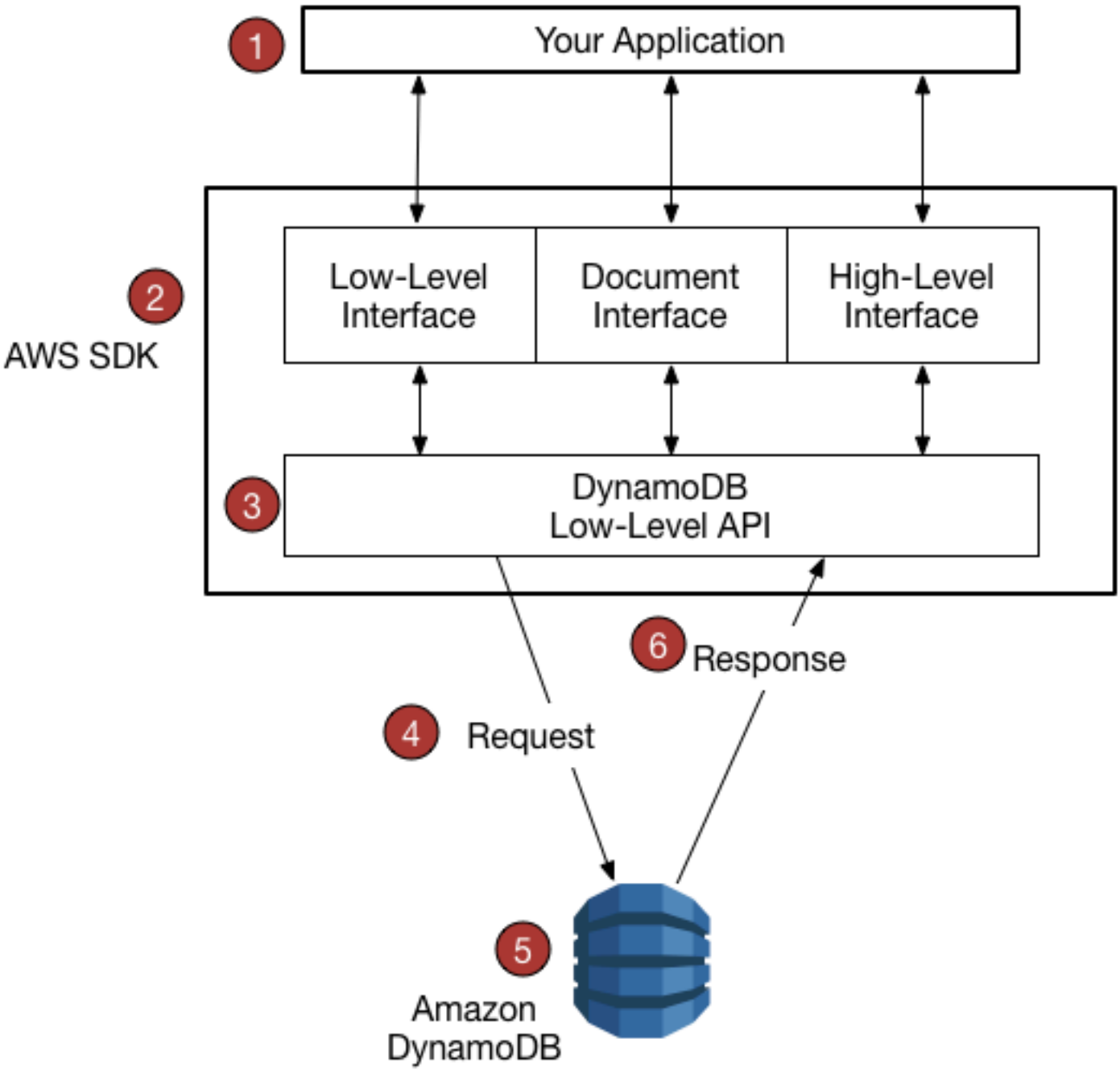
> **Note**
>
> In December 2017, AWS began the process of migrating all Amazon DynamoDB endpoints to use secure certificates issued by Amazon Trust Services (ATS). For more information, see Troubleshooting SSL/TLS connection establishment issues with DynamoDB.

Topics

- Overview of AWS SDK support for DynamoDB
- Programming Amazon DynamoDB with Python and Boto3
- Programming Amazon DynamoDB with JavaScript
- Programming DynamoDB with the AWS SDK for Java 2.x
- Error handling with DynamoDB
- Using DynamoDB with an AWS SDK

Overview of AWS SDK support for DynamoDB

The following diagram provides a high-level overview of Amazon DynamoDB application programming using the AWS SDKs.



1. You write an application using an AWS SDK for your programming language.
2. Each AWS SDK provides one or more programmatic interfaces for working with DynamoDB. The specific interfaces available depend on which programming language and AWS SDK you use. Options include:
   - Low-level interfaces that work with DynamoDB
   - Document interfaces that work with DynamoDB
   - Object persistence interfaces that work with DynamoDB

- High Level Interfaces

3. The AWS SDK constructs HTTP(S) requests for use with the low-level DynamoDB API.
4. The AWS SDK sends the request to the DynamoDB endpoint.
5. DynamoDB runs the request. If the request is successful, DynamoDB returns an HTTP 200 response code (OK). If the request is unsuccessful, DynamoDB returns an HTTP error code and an error message.
6. The AWS SDK processes the response and propagates it back to your application.

Each of the AWS SDKs provides important services to your application, including the following:

- Formatting HTTP(S) requests and serializing request parameters.
- Generating a cryptographic signature for each request.
- Forwarding requests to a DynamoDB endpoint and receiving responses from DynamoDB.
- Extracting the results from those responses.
- Implementing basic retry logic in case of errors.

You do not need to write code for any of these tasks.

> **Note**
>
> For more information about AWS SDKs, including installation instructions and documentation, see Tools for Amazon Web Services.

SDK support of AWS account-based endpoints

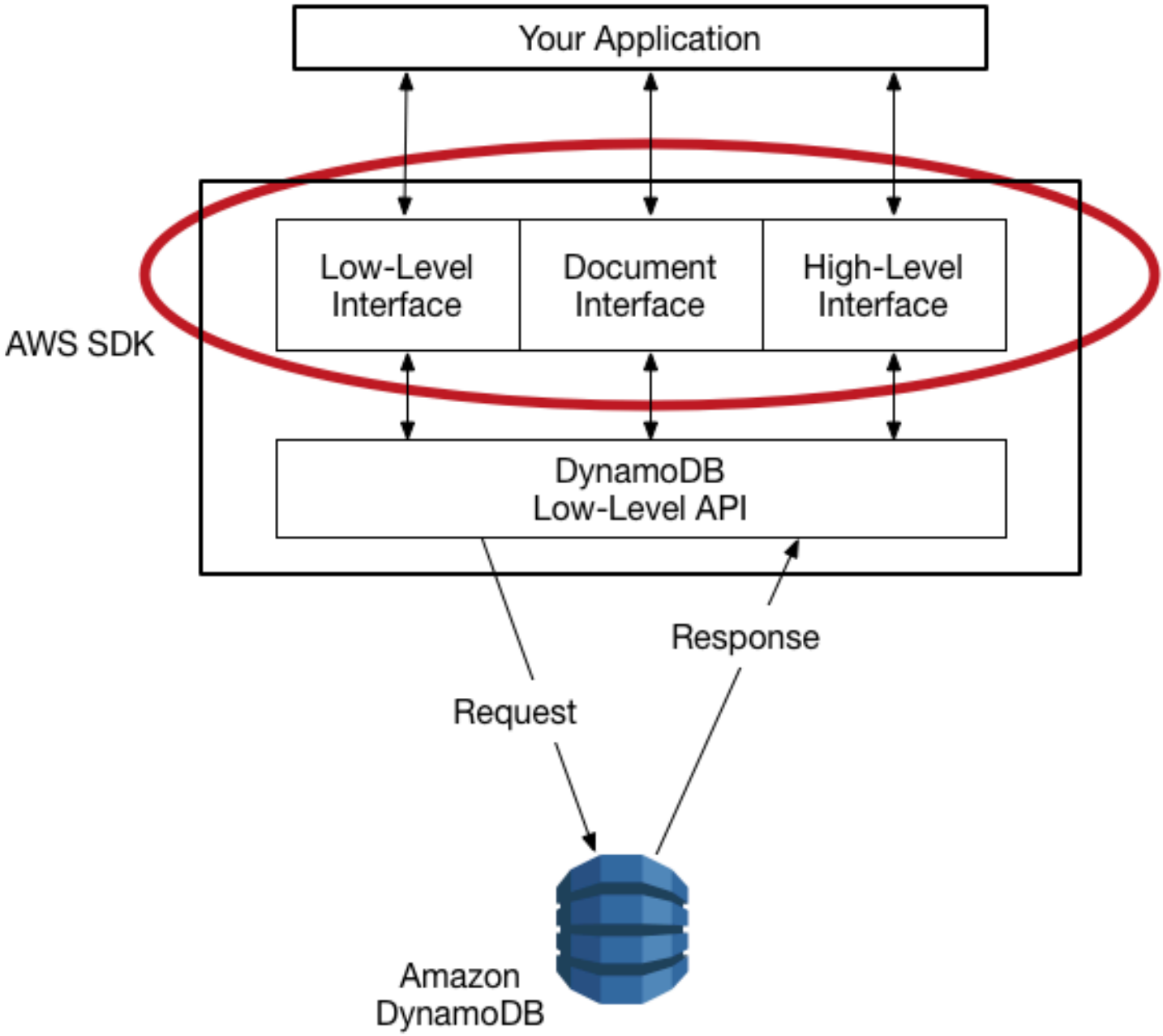
AWS is rolling out SDK support for AWS-account-based endpoints for DynamoDB, starting with the AWS SDK for Java V1 on September 4, 2024. These new endpoints help AWS to ensure high performance and scalability. The updated SDKs will automatically use the new endpoints, which have the format `https://(account-id).ddb.(region).amazonaws.com`.

If you use a single instance of an SDK client to make requests to multiple accounts, your application will have fewer opportunities to reuse connections. AWS recommends modifying your applications to connect to fewer accounts per SDK client instance. An alternative is to set your SDK client

to continue using Regional endpoints using the `ACCOUNT_ID_ENDPOINT_MODE` setting, as documented in the *AWS SDKs and Tools Reference Guide*.

Programmatic interfaces that work with DynamoDB

Every AWS SDK provides one or more programmatic interfaces for working with Amazon DynamoDB. These interfaces range from simple low-level DynamoDB wrappers to object-oriented persistence layers. The available interfaces vary depending on the AWS SDK and programming language that you use.



The following section highlights some of the interfaces available, using the AWS SDK for Java as an example. (Not all interfaces are available in all AWS SDKs.)

Topics

- Low-level interfaces that work with DynamoDB
- Document interfaces that work with DynamoDB
- Object persistence interfaces that work with DynamoDB

Low-level interfaces that work with DynamoDB

Every language-specific AWS SDK provides a low-level interface for Amazon DynamoDB, with methods that closely resemble low-level DynamoDB API requests.

In some cases, you will need to identify the data types of the attributes using Data type descriptors, such as `S` for string or `N` for number.

> **Note**
>
> A low-level interface is available in every language-specific AWS SDK.

The following Java program uses the low-level interface of the AWS SDK for Java.

Low-level interface example

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.GetItemRequest;
import java.util.HashMap;
import java.util.Map;
import java.util.Set;

/**
 * Before running this Java V2 code example, set up your development
 * environment, including your credentials.
 *
 * For more information, see the following documentation topic:
 *
```

```
 * https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-started.html
 *
 * To get an item from an Amazon DynamoDB table using the AWS SDK for Java V2,
 * its better practice to use the
 * Enhanced Client, see the EnhancedGetItem example.
 */
public class GetItem {
    public static void main(String[] args) {
        final String usage = """

                Usage:
                    <tableName> <key> <keyVal>

                Where:
                    tableName - The Amazon DynamoDB table from which an item is
 retrieved (for example, Music3).\s
                    key - The key used in the Amazon DynamoDB table (for example,
 Artist).\s
                    keyval - The key value that represents the item to get (for
 example, Famous Band).
                """;

        if (args.length != 3) {
            System.out.println(usage);
            System.exit(1);
        }

        String tableName = args[0];
        String key = args[1];
        String keyVal = args[2];
        System.out.format("Retrieving item \"%s\" from \"%s\"\n", keyVal, tableName);
        Region region = Region.US_EAST_1;
        DynamoDbClient ddb = DynamoDbClient.builder()
                .region(region)
                .build();

        getDynamoDBItem(ddb, tableName, key, keyVal);
        ddb.close();
    }

    public static void getDynamoDBItem(DynamoDbClient ddb, String tableName, String
 key, String keyVal) {
        HashMap<String, AttributeValue> keyToGet = new HashMap<>();
        keyToGet.put(key, AttributeValue.builder()
```

```
                .s(keyVal)
                .build());

        GetItemRequest request = GetItemRequest.builder()
                .key(keyToGet)
                .tableName(tableName)
                .build();

        try {
            // If there is no matching item, GetItem does not return any data.
            Map<String, AttributeValue> returnedItem = ddb.getItem(request).item();
            if (returnedItem.isEmpty())
                System.out.format("No item found with the key %s!\n", key);
            else {
                Set<String> keys = returnedItem.keySet();
                System.out.println("Amazon DynamoDB table attributes: \n");
                for (String key1 : keys) {
                    System.out.format("%s: %s\n", key1,
 returnedItem.get(key1).toString());
                }
            }

        } catch (DynamoDbException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }
}
```

Document interfaces that work with DynamoDB

Many AWS SDKs provide a document interface, allowing you to perform data plane operations (create, read, update, delete) on tables and indexes. With a document interface, you do not need to specify Data type descriptors. The data types are implied by the semantics of the data itself. These AWS SDKs also provide methods to easily convert JSON documents to and from native Amazon DynamoDB data types.

> **Note**
>
> Document interfaces are available in the AWS SDKs for Java, .NET, Node.js, and JavaScript SDK.

The following Java program uses the document interface of the AWS SDK for Java. The program creates a `Table` object that represents the `Music` table, and then asks that object to use `GetItem` to retrieve a song. The program then prints the year that the song was released.

The `software.amazon.dynamodb.document.DynamoDB` class implements the DynamoDB document interface. Note how `DynamoDB` acts as a wrapper around the low-level client (`AmazonDynamoDB`).

Document interface example

```
package com.amazonaws.codesamples.gsg;

import software.amazon.dynamodb.AmazonDynamoDB;
import software.amazon.dynamodb.AmazonDynamoDBClientBuilder;
import software.amazon.dynamodb.document.DynamoDB;
import software.amazon.dynamodb.document.GetItemOutcome;
import software.amazon.dynamodb.document.Table;

public class MusicDocumentDemo {

    public static void main(String[] args) {

        AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();
        DynamoDB docClient = new DynamoDB(client);

        Table table = docClient.getTable("Music");
        GetItemOutcome outcome = table.getItemOutcome(
                "Artist", "No One You Know",
                "SongTitle", "Call Me Today");

        int year = outcome.getItem().getInt("Year");
        System.out.println("The song was released in " + year);

    }
}
```

Object persistence interfaces that work with DynamoDB

Some AWS SDKs provide an object persistence interface where you do not directly perform data plane operations. Instead, you create objects that represent items in Amazon DynamoDB tables and indexes, and interact only with those objects. This allows you to write object-centric code, rather than database-centric code.

> **Note**
>
> Object persistence interfaces are available in the AWS SDKs for Java and .NET. For more information, see Higher-level programming interfaces for DynamoDB for DynamoDB.

Object persistence interface example

```
import com.example.dynamodb.Customer;
import software.amazon.awssdk.enhanced.dynamodb.DynamoDbEnhancedClient;
import software.amazon.awssdk.enhanced.dynamodb.DynamoDbTable;
import software.amazon.awssdk.enhanced.dynamodb.Key;
import software.amazon.awssdk.enhanced.dynamodb.TableSchema;
import software.amazon.awssdk.enhanced.dynamodb.model.GetItemEnhancedRequest;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
```

```
import com.example.dynamodb.Customer;
import software.amazon.awssdk.enhanced.dynamodb.DynamoDbEnhancedClient;
import software.amazon.awssdk.enhanced.dynamodb.DynamoDbTable;
import software.amazon.awssdk.enhanced.dynamodb.Key;
import software.amazon.awssdk.enhanced.dynamodb.TableSchema;
import software.amazon.awssdk.enhanced.dynamodb.model.GetItemEnhancedRequest;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;

/*
 * Before running this code example, create an Amazon DynamoDB table named Customer
 with these columns:
 *   - id - the id of the record that is the key. Be sure one of the id values is
 `id101`
 *   - custName - the customer name
 *   - email - the email value
 *   - registrationDate - an instant value when the item was added to the table. These
 values
 *                        need to be in the form of `YYYY-MM-DDTHH:mm:ssZ`, such as
 2022-07-11T00:00:00Z
 *
 * Also, ensure that you have set up your development environment, including your
 credentials.
```

```
 *
 * For information, see this documentation topic:
 *
 * https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-started.html
 */

public class EnhancedGetItem {
    public static void main(String[] args) {
        Region region = Region.US_EAST_1;
        DynamoDbClient ddb = DynamoDbClient.builder()
                .region(region)
                .build();

        DynamoDbEnhancedClient enhancedClient = DynamoDbEnhancedClient.builder()
                .dynamoDbClient(ddb)
                .build();

        getItem(enhancedClient);
        ddb.close();
    }

    public static String getItem(DynamoDbEnhancedClient enhancedClient) {
        Customer result = null;
        try {
            DynamoDbTable<Customer> table = enhancedClient.table("Customer",
 TableSchema.fromBean(Customer.class));
            Key key = Key.builder()
                    .partitionValue("id101").sortValue("tred@noserver.com")
                    .build();

            // Get the item by using the key.
            result = table.getItem(
                    (GetItemEnhancedRequest.Builder requestBuilder) ->
 requestBuilder.key(key));
            System.out.println("******* The description value is " +
 result.getCustName());

        } catch (DynamoDbException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
        return result.getCustName();
    }
```

```
}
```

Higher-level programming interfaces for DynamoDB

The AWS SDKs provide applications with low-level interfaces for working with Amazon DynamoDB. These client-side classes and methods correspond directly to the low-level DynamoDB API. However, many developers experience a sense of disconnect, or *impedance mismatch*, when they need to map complex data types to items in a database table. With a low-level database interface, developers must write methods for reading or writing object data to database tables, and vice versa. The amount of extra code required for each combination of object type and database table can seem overwhelming.

To simplify development, the AWS SDKs for Java and .NET provide additional interfaces with higher levels of abstraction. The higher-level interfaces for DynamoDB let you define the relationships between objects in your program and the database tables that store those objects' data. After you define this mapping, you call simple object methods such as `save`, `load`, or `delete`, and the underlying low-level DynamoDB operations are automatically invoked on your behalf. This allows you to write object-centric code, rather than database-centric code.

The higher-level programming interfaces for DynamoDB are available in the AWS SDKs for Java and .NET.

Java

- Java 1.x: DynamoDBMapper
- Java 2.x: DynamoDB Enhanced Client

.NET

- Working with the .NET document model in DynamoDB
- Working with the .NET object persistence model and DynamoDB

Java 1.x: DynamoDBMapper

> **Note**
>
> The SDK for Java has two versions: 1.x and 2.x. The end-of-support for 1.x was announced on January 12, 2024. It will and its end-of-support is due on December 31, 2025. For new development, we highly recommend that you use 2.x.

The AWS SDK for Java provides a `DynamoDBMapper` class, allowing you to map your client-side classes to Amazon DynamoDB tables. To use `DynamoDBMapper`, you define the relationship between items in a DynamoDB table and their corresponding object instances in your code. The `DynamoDBMapper` class enables you to perform various create, read, update, and delete (CRUD) operations on items, and run queries and scans against tables.

Topics

- DynamoDBMapper Class
- Supported data types for DynamoDBMapper for Java
- Java Annotations for DynamoDB
- Optional configuration settings for DynamoDBMapper
- DynamoDB and optimistic locking with version number
- Mapping arbitrary data in DynamoDB
- DynamoDBMapper examples

> **Note**
>
> The `DynamoDBMapper` class does not allow you to create, update, or delete tables. To perform those tasks, use the low-level SDK for Java interface instead.

The SDK for Java provides a set of annotation types so that you can map your classes to tables. For example, consider a `ProductCatalog` table that has `Id` as the partition key.

```
ProductCatalog(Id, ...)
```

You can map a class in your client application to the `ProductCatalog` table as shown in the following Java code. This code defines a plain old Java object (POJO) named `CatalogItem`, which uses annotations to map object fields to DynamoDB attribute names.

Example

```
package com.amazonaws.codesamples;

import java.util.Set;

import software.amazon.dynamodb.datamodeling.DynamoDBAttribute;
import software.amazon.dynamodb.datamodeling.DynamoDBHashKey;
import software.amazon.dynamodb.datamodeling.DynamoDBIgnore;
import software.amazon.dynamodb.datamodeling.DynamoDBTable;

@DynamoDBTable(tableName="ProductCatalog")
public class CatalogItem {

    private Integer id;
    private String title;
    private String ISBN;
    private Set<String> bookAuthors;
    private String someProp;

    @DynamoDBHashKey(attributeName="Id")
    public Integer getId() { return id; }
    public void setId(Integer id) {this.id = id; }

    @DynamoDBAttribute(attributeName="Title")
    public String getTitle() {return title; }
    public void setTitle(String title) { this.title = title; }

    @DynamoDBAttribute(attributeName="ISBN")
    public String getISBN() { return ISBN; }
    public void setISBN(String ISBN) { this.ISBN = ISBN; }

    @DynamoDBAttribute(attributeName="Authors")
    public Set<String> getBookAuthors() { return bookAuthors; }
    public void setBookAuthors(Set<String> bookAuthors) { this.bookAuthors =
 bookAuthors; }

    @DynamoDBIgnore
    public String getSomeProp() { return someProp; }
```

```
    public void setSomeProp(String someProp) { this.someProp = someProp; }
}
```

In the preceding code, the `@DynamoDBTable` annotation maps the `CatalogItem` class to the `ProductCatalog` table. You can store individual class instances as items in the table. In the class definition, the `@DynamoDBHashKey` annotation maps the `Id` property to the primary key.

By default, the class properties map to the same name attributes in the table. The properties `Title` and `ISBN` map to the same name attributes in the table.

The `@DynamoDBAttribute` annotation is optional when the name of the DynamoDB attribute matches the name of the property declared in the class. When they differ, use this annotation with the `attributeName` parameter to specify which DynamoDB attribute this property corresponds to.

In the preceding example, the `@DynamoDBAttribute` annotation is added to each property to ensure that the property names match exactly with the tables created in a previous step, and to be consistent with the attribute names used in other code examples in this guide.

Your class definition can have properties that don't map to any attributes in the table. You identify these properties by adding the `@DynamoDBIgnore` annotation. In the preceding example, the `SomeProp` property is marked with the `@DynamoDBIgnore` annotation. When you upload a `CatalogItem` instance to the table, your `DynamoDBMapper` instance does not include the `SomeProp` property. In addition, the mapper does not return this attribute when you retrieve an item from the table.

After you define your mapping class, you can use `DynamoDBMapper` methods to write an instance of that class to a corresponding item in the `Catalog` table. The following code example demonstrates this technique.

```
AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();

DynamoDBMapper mapper = new DynamoDBMapper(client);

CatalogItem item = new CatalogItem();
item.setId(102);
item.setTitle("Book 102 Title");
item.setISBN("222-2222222222");
item.setBookAuthors(new HashSet<String>(Arrays.asList("Author 1", "Author 2")));
item.setSomeProp("Test");
```

```
mapper.save(item);
```

The following code example shows how to retrieve the item and access some of its attributes.

```
CatalogItem partitionKey = new CatalogItem();

partitionKey.setId(102);
DynamoDBQueryExpression<CatalogItem> queryExpression = new
 DynamoDBQueryExpression<CatalogItem>()
    .withHashKeyValues(partitionKey);

List<CatalogItem> itemList = mapper.query(CatalogItem.class, queryExpression);

for (int i = 0; i < itemList.size(); i++) {
    System.out.println(itemList.get(i).getTitle());
    System.out.println(itemList.get(i).getBookAuthors());
}
```

`DynamoDBMapper` offers an intuitive, natural way of working with DynamoDB data within Java. It also provides several built-in features, such as optimistic locking, ACID transactions, autogenerated partition key and sort key values, and object versioning.

DynamoDBMapper Class

The `DynamoDBMapper` class is the entry point to Amazon DynamoDB. It provides access to a DynamoDB endpoint and enables you to access your data in various tables. It also enables you to perform various create, read, update, and delete (CRUD) operations on items, and run queries and scans against tables. This class provides the following methods for working with DynamoDB.

For the corresponding Javadoc documentation, see DynamoDBMapper in the *AWS SDK for Java API Reference*.

Topics

- save
- load
- delete
- query

- queryPage
- scan
- scanPage
- parallelScan
- batchSave
- batchLoad
- batchDelete
- batchWrite
- transactionWrite
- transactionLoad
- count
- generateCreateTableRequest
- createS3Link
- getS3ClientCache

save

Saves the specified object to the table. The object that you want to save is the only required parameter for this method. You can provide optional configuration parameters using the `DynamoDBMapperConfig` object.

If an item that has the same primary key does not exist, this method creates a new item in the table. If an item that has the same primary key exists, it updates the existing item. If the partition key and sort key are of type String and are annotated with `@DynamoDBAutoGeneratedKey`, they are given a random universally unique identifier (UUID) if left uninitialized. Version fields that are annotated with `@DynamoDBVersionAttribute` are incremented by one. Additionally, if a version field is updated or a key generated, the object passed in is updated as a result of the operation.

By default, only attributes corresponding to mapped class properties are updated. Any additional existing attributes on an item are unaffected. However, if you specify `SaveBehavior.CLOBBER`, you can force the item to be completely overwritten.

```
DynamoDBMapperConfig config = DynamoDBMapperConfig.builder()
    .withSaveBehavior(DynamoDBMapperConfig.SaveBehavior.CLOBBER).build();
```

```
mapper.save(item, config);
```

If you have versioning enabled, the client-side and server-side item versions must match. However, the version does not need to match if the `SaveBehavior.CLOBBER` option is used. For more information about versioning, see DynamoDB and optimistic locking with version number.

load

Retrieves an item from a table. You must provide the primary key of the item that you want to retrieve. You can provide optional configuration parameters using the `DynamoDBMapperConfig` object. For example, you can optionally request strongly consistent reads to ensure that this method retrieves only the latest item values as shown in the following Java statement.

```
DynamoDBMapperConfig config = DynamoDBMapperConfig.builder()
    .withConsistentReads(DynamoDBMapperConfig.ConsistentReads.CONSISTENT).build();

CatalogItem item = mapper.load(CatalogItem.class, item.getId(), config);
```

By default, DynamoDB returns the item that has values that are eventually consistent. For information about the eventual consistency model of DynamoDB, see DynamoDB read consistency.

delete

Deletes an item from the table. You must pass in an object instance of the mapped class.

If you have versioning enabled, the client-side and server-side item versions must match. However, the version does not need to match if the `SaveBehavior.CLOBBER` option is used. For more information about versioning, see DynamoDB and optimistic locking with version number.

query

Queries a table or a secondary index.

Assume that you have a table, `Reply`, that stores forum thread replies. Each thread subject can have zero or more replies. The primary key of the `Reply` table consists of the `Id` and `ReplyDateTime` fields, where `Id` is the partition key and `ReplyDateTime` is the sort key of the primary key.

```
Reply ( Id, ReplyDateTime, ... )
```

Assume that you created a mapping between a `Reply` class and the corresponding `Reply` table in DynamoDB. The following Java code uses `DynamoDBMapper` to find all replies in the past two weeks for a specific thread subject.

Example

```
String forumName = "&DDB;";
String forumSubject = "&DDB; Thread 1";
String partitionKey = forumName + "#" + forumSubject;

long twoWeeksAgoMilli = (new Date()).getTime() - (14L*24L*60L*60L*1000L);
Date twoWeeksAgo = new Date();
twoWeeksAgo.setTime(twoWeeksAgoMilli);
SimpleDateFormat df = new SimpleDateFormat("yyyy-MM-dd'T'HH:mm:ss.SSS'Z'");
String twoWeeksAgoStr = df.format(twoWeeksAgo);

Map<String, AttributeValue> eav = new HashMap<String, AttributeValue>();
eav.put(":v1", new AttributeValue().withS(partitionKey));
eav.put(":v2",new AttributeValue().withS(twoWeeksAgoStr.toString()));

DynamoDBQueryExpression<Reply> queryExpression = new DynamoDBQueryExpression<Reply>()
    .withKeyConditionExpression("Id = :v1 and ReplyDateTime > :v2")
    .withExpressionAttributeValues(eav);

List<Reply> latestReplies = mapper.query(Reply.class, queryExpression);
```

The query returns a collection of `Reply` objects.

By default, the `query` method returns a "lazy-loaded" collection. It initially returns only one page of results, and then makes a service call for the next page if needed. To obtain all the matching items, iterate over the `latestReplies` collection.

Note that calling the `size()` method on the collection will load every result in order to provide an accurate count. This can result in a lot of provisioned throughput being consumed, and on a very large table could even exhaust all the memory in your JVM.

To query an index, you must first model the index as a mapper class. Suppose that the `Reply` table has a global secondary index named *PostedBy-Message-Index*. The partition key for this index is `PostedBy`, and the sort key is `Message`. The class definition for an item in the index would look like the following.

```
@DynamoDBTable(tableName="Reply")
```

```
public class PostedByMessage {
    private String postedBy;
    private String message;

    @DynamoDBIndexHashKey(globalSecondaryIndexName = "PostedBy-Message-Index",
 attributeName = "PostedBy")
    public String getPostedBy() { return postedBy; }
    public void setPostedBy(String postedBy) { this.postedBy = postedBy; }

    @DynamoDBIndexRangeKey(globalSecondaryIndexName = "PostedBy-Message-Index",
 attributeName = "Message")
    public String getMessage() { return message; }
    public void setMessage(String message) { this.message = message; }

   // Additional properties go here.
}
```

The `@DynamoDBTable` annotation indicates that this index is associated with the `Reply` table. The `@DynamoDBIndexHashKey` annotation denotes the partition key (*PostedBy*) of the index, and `@DynamoDBIndexRangeKey` denotes the sort key (*Message*) of the index.

Now you can use `DynamoDBMapper` to query the index, retrieving a subset of messages that were posted by a particular user. You do not need to specify the index name if you do not have conflicting mappings across tables and indexes and the mappings are already made in the mapper. The mapper will infer based on the primary key and sort key. The following code queries a global secondary index. Because global secondary indexes support eventually consistent reads but not strongly consistent reads, you must specify `withConsistentRead(false)`.

```
HashMap<String, AttributeValue> eav = new HashMap<String, AttributeValue>();
eav.put(":v1",  new AttributeValue().withS("User A"));
eav.put(":v2",  new AttributeValue().withS("DynamoDB"));

DynamoDBQueryExpression<PostedByMessage> queryExpression = new
 DynamoDBQueryExpression<PostedByMessage>()
    .withIndexName("PostedBy-Message-Index")
    .withConsistentRead(false)
    .withKeyConditionExpression("PostedBy = :v1 and begins_with(Message, :v2)")
    .withExpressionAttributeValues(eav);

List<PostedByMessage> iList =  mapper.query(PostedByMessage.class, queryExpression);
```

The query returns a collection of `PostedByMessage` objects.

queryPage

Queries a table or secondary index and returns a single page of matching results. As with the `query` method, you must specify a partition key value and a query filter that is applied on the sort key attribute. However, `queryPage` returns only the first "page" of data, that is, the amount of data that fits in 1 MB

scan

Scans an entire table or a secondary index. You can optionally specify a `FilterExpression` to filter the result set.

Assume that you have a table, `Reply`, that stores forum thread replies. Each thread subject can have zero or more replies. The primary key of the `Reply` table consists of the `Id` and `ReplyDateTime` fields, where `Id` is the partition key and `ReplyDateTime` is the sort key of the primary key.

```
Reply ( Id, ReplyDateTime, ... )
```

If you mapped a Java class to the `Reply` table, you can use the `DynamoDBMapper` to scan the table. For example, the following Java code scans the entire `Reply` table, returning only the replies for a particular year.

Example

```
HashMap<String, AttributeValue> eav = new HashMap<String, AttributeValue>();
eav.put(":v1", new AttributeValue().withS("2015"));

DynamoDBScanExpression scanExpression = new DynamoDBScanExpression()
    .withFilterExpression("begins_with(ReplyDateTime,:v1)")
    .withExpressionAttributeValues(eav);

List<Reply> replies =  mapper.scan(Reply.class, scanExpression);
```

By default, the `scan` method returns a "lazy-loaded" collection. It initially returns only one page of results, and then makes a service call for the next page if needed. To obtain all the matching items, iterate over the `replies` collection.

Note that calling the `size()` method on the collection will load every result in order to provide an accurate count. This can result in a lot of provisioned throughput being consumed, and on a very large table could even exhaust all the memory in your JVM.

To scan an index, you must first model the index as a mapper class. Suppose that the `Reply` table has a global secondary index named `PostedBy-Message-Index`. The partition key for this index is `PostedBy`, and the sort key is `Message`. A mapper class for this index is shown in the query section. It uses the `@DynamoDBIndexHashKey` and `@DynamoDBIndexRangeKey` annotations to specify the index partition key and sort key.

The following code example scans `PostedBy-Message-Index`. It does not use a scan filter, so all of the items in the index are returned to you.

```
DynamoDBScanExpression scanExpression = new DynamoDBScanExpression()
    .withIndexName("PostedBy-Message-Index")
    .withConsistentRead(false);

    List<PostedByMessage> iList =  mapper.scan(PostedByMessage.class, scanExpression);
    Iterator<PostedByMessage> indexItems = iList.iterator();
```

scanPage

Scans a table or secondary index and returns a single page of matching results. As with the `scan` method, you can optionally specify a `FilterExpression` to filter the result set. However, `scanPage` only returns the first "page" of data, that is, the amount of data that fits within 1 MB.

parallelScan

Performs a parallel scan of an entire table or secondary index. You specify a number of logical segments for the table, along with a scan expression to filter the results. The `parallelScan` divides the scan task among multiple workers, one for each logical segment; the workers process the data in parallel and return the results.

The following Java code example performs a parallel scan on the `Product` table.

```
int numberOfThreads = 4;

Map<String, AttributeValue> eav = new HashMap<String, AttributeValue>();
eav.put(":n", new AttributeValue().withN("100"));

DynamoDBScanExpression scanExpression = new DynamoDBScanExpression()
    .withFilterExpression("Price <= :n")
    .withExpressionAttributeValues(eav);
```

```
List<Product> scanResult = mapper.parallelScan(Product.class, scanExpression,
 numberOfThreads);
```

batchSave

Saves objects to one or more tables using one or more calls to the `AmazonDynamoDB.batchWriteItem` method. This method does not provide transaction guarantees.

The following Java code saves two items (books) to the `ProductCatalog` table.

```
Book book1 = new Book();
book1.setId(901);
book1.setProductCategory("Book");
book1.setTitle("Book 901 Title");

Book book2 = new Book();
book2.setId(902);
book2.setProductCategory("Book");
book2.setTitle("Book 902 Title");

mapper.batchSave(Arrays.asList(book1, book2));
```

batchLoad

Retrieves multiple items from one or more tables using their primary keys.

The following Java code retrieves two items from two different tables.

```
ArrayList<Object> itemsToGet = new ArrayList<Object>();

ForumItem forumItem = new ForumItem();
forumItem.setForumName("Amazon DynamoDB");
itemsToGet.add(forumItem);

ThreadItem threadItem = new ThreadItem();
threadItem.setForumName("Amazon DynamoDB");
threadItem.setSubject("Amazon DynamoDB thread 1 message text");
itemsToGet.add(threadItem);

Map<String, List<Object>> items = mapper.batchLoad(itemsToGet);
```

batchDelete

Deletes objects from one or more tables using one or more calls to the `AmazonDynamoDB.batchWriteItem` method. This method does not provide transaction guarantees.

The following Java code deletes two items (books) from the `ProductCatalog` table.

```
Book book1 = mapper.load(Book.class, 901);
Book book2 = mapper.load(Book.class, 902);
mapper.batchDelete(Arrays.asList(book1, book2));
```

batchWrite

Saves objects to and deletes objects from one or more tables using one or more calls to the `AmazonDynamoDB.batchWriteItem` method. This method does not provide transaction guarantees or support versioning (conditional puts or deletes).

The following Java code writes a new item to the `Forum` table, writes a new item to the `Thread` table, and deletes an item from the `ProductCatalog` table.

```
// Create a Forum item to save
Forum forumItem = new Forum();
forumItem.setName("Test BatchWrite Forum");

// Create a Thread item to save
Thread threadItem = new Thread();
threadItem.setForumName("AmazonDynamoDB");
threadItem.setSubject("My sample question");

// Load a ProductCatalog item to delete
Book book3 = mapper.load(Book.class, 903);

List<Object> objectsToWrite = Arrays.asList(forumItem, threadItem);
List<Book> objectsToDelete = Arrays.asList(book3);

mapper.batchWrite(objectsToWrite, objectsToDelete);
```

transactionWrite

Saves objects to and deletes objects from one or more tables using one call to the `AmazonDynamoDB.transactWriteItems` method.

For a list of transaction-specific exceptions, see TransactWriteItems errors.

For more information about DynamoDB transactions and the provided atomicity, consistency, isolation, and durability (ACID) guarantees see Amazon DynamoDB Transactions.

> **Note**
>
> This method does not support the following:
>
> - DynamoDBMapperConfig.SaveBehavior.

The following Java code writes a new item to each of the `Forum` and `Thread` tables, transactionally.

```
Thread s3ForumThread = new Thread();
s3ForumThread.setForumName("S3 Forum");
s3ForumThread.setSubject("Sample Subject 1");
s3ForumThread.setMessage("Sample Question 1");

Forum s3Forum = new Forum();
s3Forum.setName("S3 Forum");
s3Forum.setCategory("Amazon Web Services");
s3Forum.setThreads(1);

TransactionWriteRequest transactionWriteRequest = new TransactionWriteRequest();
transactionWriteRequest.addPut(s3Forum);
transactionWriteRequest.addPut(s3ForumThread);
mapper.transactionWrite(transactionWriteRequest);
```

transactionLoad

Loads objects from one or more tables using one call to the `AmazonDynamoDB.transactGetItems` method.

For a list of transaction-specific exceptions, see TransactGetItems errors.

For more information about DynamoDB transactions and the provided atomicity, consistency, isolation, and durability (ACID) guarantees see Amazon DynamoDB Transactions.

The following Java code loads one item from each of the `Forum` and `Thread` tables, transactionally.

```
Forum dynamodbForum = new Forum();
dynamodbForum.setName("DynamoDB Forum");
Thread dynamodbForumThread = new Thread();
dynamodbForumThread.setForumName("DynamoDB Forum");

TransactionLoadRequest transactionLoadRequest = new TransactionLoadRequest();
transactionLoadRequest.addLoad(dynamodbForum);
transactionLoadRequest.addLoad(dynamodbForumThread);
mapper.transactionLoad(transactionLoadRequest);
```

count

Evaluates the specified scan expression and returns the count of matching items. No item data is returned.

generateCreateTableRequest

Parses a POJO class that represents a DynamoDB table, and returns a `CreateTableRequest` for that table.

createS3Link

Creates a link to an object in Amazon S3. You must specify a bucket name and a key name, which uniquely identifies the object in the bucket.

To use `createS3Link`, your mapper class must define getter and setter methods. The following code example illustrates this by adding a new attribute and getter/setter methods to the `CatalogItem` class.

```
@DynamoDBTable(tableName="ProductCatalog")
public class CatalogItem {

    ...

    public S3Link productImage;

    ....

    @DynamoDBAttribute(attributeName = "ProductImage")
    public S3Link getProductImage() {
            return productImage;
    }
```

```
    public void setProductImage(S3Link productImage) {
        this.productImage = productImage;
    }

...
}
```

The following Java code defines a new item to be written to the `Product` table. The item includes a link to a product image; the image data is uploaded to Amazon S3.

```
CatalogItem item = new CatalogItem();

item.setId(150);
item.setTitle("Book 150 Title");

String amzn-s3-demo-bucket = "amzn-s3-demo-bucket";
String myS3Key = "productImages/book_150_cover.jpg";
item.setProductImage(mapper.createS3Link(amzn-s3-demo-bucket, myS3Key));

item.getProductImage().uploadFrom(new File("/file/path/book_150_cover.jpg"));

mapper.save(item);
```

The `S3Link` class provides many other methods for manipulating objects in Amazon S3. For more information, see the Javadocs for `S3Link`.

getS3ClientCache

Returns the underlying `S3ClientCache` for accessing Amazon S3. An `S3ClientCache` is a smart Map for `AmazonS3Client` objects. If you have multiple clients, an `S3ClientCache` can help you keep the clients organized by AWS Region, and can create new Amazon S3 clients on demand.

Supported data types for DynamoDBMapper for Java

This section describes the supported primitive Java data types, collections, and arbitrary data types in Amazon DynamoDB.

Amazon DynamoDB supports the following primitive Java data types and primitive wrapper classes.

- `String`
- `Boolean`, `boolean`

- `Byte`, `byte`
- `Date` (as ISO_8601 millisecond-precision string, shifted to UTC)
- `Calendar` (as ISO_8601 millisecond-precision string, shifted to UTC)
- `Long`, `long`
- `Integer`, `int`
- `Double`, `double`
- `Float`, `float`
- `BigDecimal`
- `BigInteger`

Note

- For more information about DynamoDB naming rules and the various supported data types, see Supported data types and naming rules in Amazon DynamoDB.
- Empty Binary values are supported by the DynamoDBMapper.
- Empty String values are supported by AWS SDK for Java 2.x.

  In AWS SDK for Java 1.x, DynamoDBMapper supports reading of empty String attribute values, however, it will not write empty String attribute values because these attributes are dropped from the request.

DynamoDB supports the Java Set, List, and Map collection types. The following table summarizes how these Java types map to the DynamoDB types.

Java type	DynamoDB type
All number types	N (number type)
Strings	S (string type)
Boolean	BOOL (Boolean type), 0 or 1.
ByteBuffer	B (binary type)

Java type	DynamoDB type
Date	S (string type). The Date values are stored as ISO-8601 formatted strings.
Set collection types	SS (string set) type, NS (number set) type, or BS (binary set) type.

The `DynamoDBTypeConverter` interface lets you map your own arbitrary data types to a data type that is natively supported by DynamoDB. For more information, see Mapping arbitrary data in DynamoDB.

Java Annotations for DynamoDB

This section describes the annotations that are available for mapping your classes and properties to tables and attributes in Amazon DynamoDB.

For the corresponding Javadoc documentation, see Annotation Types Summary in the AWS SDK for Java API Reference.

> **Note**
>
> In the following annotations, only `DynamoDBTable` and the `DynamoDBHashKey` are required.

Topics

- DynamoDBAttribute
- DynamoDBAutoGeneratedKey
- DynamoDBAutoGeneratedTimestamp
- DynamoDBDocument
- DynamoDBHashKey
- DynamoDBIgnore
- DynamoDBIndexHashKey
- DynamoDBIndexRangeKey
- DynamoDBRangeKey

- DynamoDBTable
- DynamoDBTypeConverted
- DynamoDBTyped
- DynamoDBVersionAttribute

DynamoDBAttribute

Maps a property to a table attribute. By default, each class property maps to an item attribute with the same name. However, if the names are not the same, you can use this annotation to map a property to the attribute. In the following Java snippet, the `DynamoDBAttribute` maps the `BookAuthors` property to the `Authors` attribute name in the table.

```
@DynamoDBAttribute(attributeName = "Authors")
public List<String> getBookAuthors() { return BookAuthors; }
public void setBookAuthors(List<String> BookAuthors) { this.BookAuthors =
 BookAuthors; }
```

The `DynamoDBMapper` uses `Authors` as the attribute name when saving the object to the table.

DynamoDBAutoGeneratedKey

Marks a partition key or sort key property as being autogenerated. `DynamoDBMapper` generates a random UUID when saving these attributes. Only String properties can be marked as autogenerated keys.

The following example demonstrates using autogenerated keys.

```
@DynamoDBTable(tableName="AutoGeneratedKeysExample")
public class AutoGeneratedKeys {
    private String id;
    private String payload;

    @DynamoDBHashKey(attributeName = "Id")
    @DynamoDBAutoGeneratedKey
    public String getId() { return id; }
    public void setId(String id) { this.id = id; }

    @DynamoDBAttribute(attributeName="payload")
    public String getPayload() { return this.payload; }
```

```
    public void setPayload(String payload) { this.payload = payload; }

    public static void saveItem() {
        AutoGeneratedKeys obj = new AutoGeneratedKeys();
        obj.setPayload("abc123");

        // id field is null at this point
        DynamoDBMapper mapper = new DynamoDBMapper(dynamoDBClient);
        mapper.save(obj);

        System.out.println("Object was saved with id " + obj.getId());
    }
}
```

DynamoDBAutoGeneratedTimestamp

Automatically generates a timestamp.

```
@DynamoDBAutoGeneratedTimestamp(strategy=DynamoDBAutoGenerateStrategy.ALWAYS)
public Date getLastUpdatedDate() { return lastUpdatedDate; }
public void setLastUpdatedDate(Date lastUpdatedDate) { this.lastUpdatedDate =
 lastUpdatedDate; }
```

Optionally, the auto-generation strategy can be defined by providing a strategy attribute. The default is `ALWAYS`.

DynamoDBDocument

Indicates that a class can be serialized as an Amazon DynamoDB document.

For example, suppose that you wanted to map a JSON document to a DynamoDB attribute of type Map (`M`). The following code example defines an item containing a nested attribute (Pictures) of type Map.

```
public class ProductCatalogItem {

    private Integer id;  //partition key
    private Pictures pictures;
    /* ...other attributes omitted... */

    @DynamoDBHashKey(attributeName="Id")
    public Integer getId() { return id;}
```

```
    public void setId(Integer id) {this.id = id;}

    @DynamoDBAttribute(attributeName="Pictures")
    public Pictures getPictures() { return pictures;}
    public void setPictures(Pictures pictures) {this.pictures = pictures;}

    // Additional properties go here.

    @DynamoDBDocument
    public static class Pictures {
        private String frontView;
        private String rearView;
        private String sideView;

        @DynamoDBAttribute(attributeName = "FrontView")
        public String getFrontView() { return frontView; }
        public void setFrontView(String frontView) { this.frontView = frontView; }

        @DynamoDBAttribute(attributeName = "RearView")
        public String getRearView() { return rearView; }
        public void setRearView(String rearView) { this.rearView = rearView; }

        @DynamoDBAttribute(attributeName = "SideView")
        public String getSideView() { return sideView; }
        public void setSideView(String sideView) { this.sideView = sideView; }

     }
}
```

You could then save a new `ProductCatalog` item, with `Pictures`, as shown in the following example.

```
ProductCatalogItem item = new ProductCatalogItem();

Pictures pix = new Pictures();
pix.setFrontView("http://example.com/products/123_front.jpg");
pix.setRearView("http://example.com/products/123_rear.jpg");
pix.setSideView("http://example.com/products/123_left_side.jpg");
item.setPictures(pix);

item.setId(123);

mapper.save(item);
```

The resulting ProductCatalog item would look like the following (in JSON format).

```
{
  "Id" : 123
  "Pictures" : {
    "SideView" : "http://example.com/products/123_left_side.jpg",
    "RearView" : "http://example.com/products/123_rear.jpg",
    "FrontView" : "http://example.com/products/123_front.jpg"
  }
}
```

DynamoDBHashKey

Maps a class property to the partition key of the table. The property must be one of the scalar string, number, or binary types. The property can't be a collection type.

Assume that you have a table, ProductCatalog, that has Id as the primary key. The following Java code defines a CatalogItem class and maps its Id property to the primary key of the ProductCatalog table using the @DynamoDBHashKey tag.

```
@DynamoDBTable(tableName="ProductCatalog")
public class CatalogItem {
    private Integer Id;
   @DynamoDBHashKey(attributeName="Id")
   public Integer getId() {
        return Id;
   }
   public void setId(Integer Id) {
        this.Id = Id;
   }
   // Additional properties go here.
}
```

DynamoDBIgnore

Indicates to the DynamoDBMapper instance that the associated property should be ignored. When saving data to the table, the DynamoDBMapper does not save this property to the table.

Applied to the getter method or the class field for a non-modeled property. If the annotation is applied directly to the class field, the corresponding getter and setter must be declared in the same class.

DynamoDBIndexHashKey

Maps a class property to the partition key of a global secondary index. The property must be one of the scalar string, number, or binary types. The property can't be a collection type.

Use this annotation if you need to `Query` a global secondary index. You must specify the index name (`globalSecondaryIndexName`). If the name of the class property is different from the index partition key, you also must specify the name of that index attribute (`attributeName`).

DynamoDBIndexRangeKey

Maps a class property to the sort key of a global secondary index or a local secondary index. The property must be one of the scalar string, number, or binary types. The property can't be a collection type.

Use this annotation if you need to `Query` a local secondary index or a global secondary index and want to refine your results using the index sort key. You must specify the index name (either `globalSecondaryIndexName` or `localSecondaryIndexName`). If the name of the class property is different from the index sort key, you must also specify the name of that index attribute (`attributeName`).

DynamoDBRangeKey

Maps a class property to the sort key of the table. The property must be one of the scalar string, number, or binary types. It cannot be a collection type.

If the primary key is composite (partition key and sort key), you can use this tag to map your class field to the sort key. For example, assume that you have a `Reply` table that stores replies for forum threads. Each thread can have many replies. So the primary key of this table is both the `ThreadId` and `ReplyDateTime`. The `ThreadId` is the partition key, and `ReplyDateTime` is the sort key.

The following Java code defines a `Reply` class and maps it to the `Reply` table. It uses both the `@DynamoDBHashKey` and `@DynamoDBRangeKey` tags to identify class properties that map to the primary key.

```
@DynamoDBTable(tableName="Reply")
public class Reply {
    private Integer id;
    private String replyDateTime;

    @DynamoDBHashKey(attributeName="Id")
    public Integer getId() { return id; }
```

```
    public void setId(Integer id) { this.id = id; }

    @DynamoDBRangeKey(attributeName="ReplyDateTime")
    public String getReplyDateTime() { return replyDateTime; }
    public void setReplyDateTime(String replyDateTime) { this.replyDateTime =
 replyDateTime; }

   // Additional properties go here.
}
```

DynamoDBTable

Identifies the target table in DynamoDB. For example, the following Java code defines a class `Developer` and maps it to the `People` table in DynamoDB.

```
@DynamoDBTable(tableName="People")
public class Developer { ...}
```

The `@DynamoDBTable` annotation can be inherited. Any new class that inherits from the `Developer` class also maps to the `People` table. For example, assume that you create a `Lead` class that inherits from the `Developer` class. Because you mapped the `Developer` class to the `People` table, the `Lead` class objects are also stored in the same table.

The `@DynamoDBTable` can also be overridden. Any new class that inherits from the `Developer` class by default maps to the same `People` table. However, you can override this default mapping. For example, if you create a class that inherits from the `Developer` class, you can explicitly map it to another table by adding the `@DynamoDBTable` annotation as shown in the following Java code example.

```
@DynamoDBTable(tableName="Managers")
public class Manager extends Developer { ...}
```

DynamoDBTypeConverted

An annotation to mark a property as using a custom type converter. Can be annotated on a user-defined annotation to pass additional properties to the `DynamoDBTypeConverter`.

The `DynamoDBTypeConverter` interface lets you map your own arbitrary data types to a data type that is natively supported by DynamoDB. For more information, see Mapping arbitrary data in DynamoDB.

DynamoDBTyped

An annotation to override the standard attribute type binding. Standard types do not require the annotation if applying the default attribute binding for that type.

DynamoDBVersionAttribute

Identifies a class property for storing an optimistic locking version number. `DynamoDBMapper` assigns a version number to this property when it saves a new item, and increments it each time you update the item. Only number scalar types are supported. For more information about data types, see Data types. For more information about versioning, see DynamoDB and optimistic locking with version number.

Optional configuration settings for DynamoDBMapper

When you create an instance of `DynamoDBMapper`, it has certain default behaviors; you can override these defaults by using the `DynamoDBMapperConfig` class.

The following code snippet creates a `DynamoDBMapper` with custom settings:

```
AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();

DynamoDBMapperConfig mapperConfig = DynamoDBMapperConfig.builder()
        .withSaveBehavior(DynamoDBMapperConfig.SaveBehavior.CLOBBER)
        .withConsistentReads(DynamoDBMapperConfig.ConsistentReads.CONSISTENT)
        .withTableNameOverride(null)

 .withPaginationLoadingStrategy(DynamoDBMapperConfig.PaginationLoadingStrategy.EAGER_LOADING)
    .build();

DynamoDBMapper mapper = new DynamoDBMapper(client, mapperConfig);
```

For more information, see *DynamoDBMapperConfig* in the AWS SDK for Java API Reference.

You can use the following arguments for an instance of `DynamoDBMapperConfig`:

- A `DynamoDBMapperConfig.ConsistentReads` enumeration value:
  - `EVENTUAL`—the mapper instance uses an eventually consistent read request.
  - `CONSISTENT`—the mapper instance uses a strongly consistent read request. You can use this optional setting with `load`, `query`, or `scan` operations. Strongly consistent reads have

implications for performance and billing; see the DynamoDB product detail page for more information.

If you do not specify a read consistency setting for your mapper instance, the default is `EVENTUAL`.

> **Note**
>
> This value is applied in the `query`, `querypage`, `load`, and `batch load` operations of the DynamoDBMapper.

- A `DynamoDBMapperConfig.PaginationLoadingStrategy` enumeration value—Controls how the mapper instance processes a paginated list of data, such as the results from a `query` or `scan`:
  - `LAZY_LOADING`—the mapper instance loads data when possible, and keeps all loaded results in memory.
  - `EAGER_LOADING`—the mapper instance loads the data as soon as the list is initialized.
  - `ITERATION_ONLY`—you can only use an Iterator to read from the list. During the iteration, the list will clear all the previous results before loading the next page, so that the list will keep at most one page of the loaded results in memory. This also means the list can only be iterated once. This strategy is recommended when handling large items, in order to reduce memory overhead.

  If you do not specify a pagination loading strategy for your mapper instance, the default is `LAZY_LOADING`.
- A `DynamoDBMapperConfig.SaveBehavior` enumeration value - Specifies how the mapper instance should deal with attributes during save operations:
  - `UPDATE`—during a save operation, all modeled attributes are updated, and unmodeled attributes are unaffected. Primitive number types (byte, int, long) are set to 0. Object types are set to null.
  - `CLOBBER`—clears and replaces all attributes, included unmodeled ones, during a save operation. This is done by deleting the item and re-creating it. Versioned field constraints are also disregarded.

  If you do not specify the save behavior for your mapper instance, the default is `UPDATE`.

> **Note**
>
> DynamoDBMapper transactional operations do not support `DynamoDBMapperConfig.SaveBehavior` enumeration.

- A `DynamoDBMapperConfig.TableNameOverride` object—Instructs the mapper instance to ignore the table name specified by a class's `DynamoDBTable` annotation, and instead use a different table name that you supply. This is useful when partitioning your data into multiple tables at runtime.

You can override the default configuration object for `DynamoDBMapper` per operation, as needed.

DynamoDB and optimistic locking with version number

Optimistic locking is a strategy to ensure that the client-side item that you are updating (or deleting) is the same as the item in Amazon DynamoDB. If you use this strategy, your database writes are protected from being overwritten by the writes of others, and vice versa.

With optimistic locking, each item has an attribute that acts as a version number. If you retrieve an item from a table, the application records the version number of that item. You can update the item, but only if the version number on the server side has not changed. If there is a version mismatch, it means that someone else has modified the item before you did. The update attempt fails, because you have a stale version of the item. If this happens, try again by retrieving the item and then trying to update it. Optimistic locking prevents you from accidentally overwriting changes that were made by others. It also prevents others from accidentally overwriting your changes.

While you can implement your own optimistic locking strategy, the AWS SDK for Java provides the `@DynamoDBVersionAttribute` annotation. In the mapping class for your table, you designate one property to store the version number, and mark it using this annotation. When you save an object, the corresponding item in the DynamoDB table will have an attribute that stores the version number. The `DynamoDBMapper` assigns a version number when you first save the object, and it automatically increments the version number each time you update the item. Your update or delete requests succeed only if the client-side object version matches the corresponding version number of the item in the DynamoDB table.

`ConditionalCheckFailedException` is thrown if:

- You use optimistic locking with `@DynamoDBVersionAttribute` and the version value on the server is different from the value on the client side.
- You specify your own conditional constraints while saving data by using `DynamoDBMapper` with `DynamoDBSaveExpression` and these constraints failed.

> **Note**
>
> - DynamoDB global tables use a "last writer wins" reconciliation between concurrent updates. If you use global tables, last writer policy wins. So in this case, the locking strategy does not work as expected.
> - `DynamoDBMapper` transactional write operations do not support `@DynamoDBVersionAttribute` annotation and condition expressions on the same object. If an object within a transactional write is annotated with `@DynamoDBVersionAttribute` and also has a condition expression, then an SdkClientException will be thrown.

For example, the following Java code defines a `CatalogItem` class that has several properties. The `Version` property is tagged with the `@DynamoDBVersionAttribute` annotation.

Example

```
@DynamoDBTable(tableName="ProductCatalog")
public class CatalogItem {

    private Integer id;
    private String title;
    private String ISBN;
    private Set<String> bookAuthors;
    private String someProp;
    private Long version;

    @DynamoDBHashKey(attributeName="Id")
    public Integer getId() { return id; }
    public void setId(Integer Id) { this.id = Id; }

    @DynamoDBAttribute(attributeName="Title")
    public String getTitle() { return title; }
    public void setTitle(String title) { this.title = title; }
```

```
    @DynamoDBAttribute(attributeName="ISBN")
    public String getISBN() { return ISBN; }
    public void setISBN(String ISBN) { this.ISBN = ISBN;}

    @DynamoDBAttribute(attributeName = "Authors")
    public Set<String> getBookAuthors() { return bookAuthors; }
    public void setBookAuthors(Set<String> bookAuthors) { this.bookAuthors =
 bookAuthors; }

    @DynamoDBIgnore
    public String getSomeProp() { return someProp;}
    public void setSomeProp(String someProp) {this.someProp = someProp;}

    @DynamoDBVersionAttribute
    public Long getVersion() { return version; }
    public void setVersion(Long version) { this.version = version;}
}
```

You can apply the `@DynamoDBVersionAttribute` annotation to nullable types provided by the primitive wrappers classes that provide a nullable type, such as `Long` and `Integer`.

Optimistic locking has the following impact on these `DynamoDBMapper` methods:

- `save` — For a new item, the `DynamoDBMapper` assigns an initial version number of 1. If you retrieve an item, update one or more of its properties, and attempt to save the changes, the save operation succeeds only if the version number on the client side and the server side match. The `DynamoDBMapper` increments the version number automatically.
- `delete` — The `delete` method takes an object as a parameter, and the `DynamoDBMapper` performs a version check before deleting the item. The version check can be disabled if `DynamoDBMapperConfig.SaveBehavior.CLOBBER` is specified in the request.

  The internal implementation of optimistic locking within `DynamoDBMapper` uses conditional update and conditional delete support provided by DynamoDB.
- `transactionWrite` —
  - `Put` — For a new item, the `DynamoDBMapper` assigns an initial version number of 1. If you retrieve an item, update one or more of its properties, and attempt to save the changes, the put operation succeeds only if the version number on the client side and the server side match. The `DynamoDBMapper` increments the version number automatically.

- `Update` — For a new item, the `DynamoDBMapper` assigns an initial version number of 1. If you retrieve an item, update one or more of its properties, and attempt to save the changes, the update operation succeeds only if the version number on the client side and the server side match. The `DynamoDBMapper` increments the version number automatically.
- `Delete` — The `DynamoDBMapper` performs a version check before deleting the item. The delete operation succeeds only if the version number on the client side and the server side match.
- `ConditionCheck` — The `@DynamoDBVersionAttribute` annotation is not supported for `ConditionCheck` operations. An SdkClientException will be thrown when a `ConditionCheck` item is annotated with `@DynamoDBVersionAttribute`.

Disabling optimistic locking

To disable optimistic locking, you can change the `DynamoDBMapperConfig.SaveBehavior` enumeration value from `UPDATE` to `CLOBBER`. You can do this by creating a `DynamoDBMapperConfig` instance that skips version checking and use this instance for all your requests. For information about `DynamoDBMapperConfig.SaveBehavior` and other optional `DynamoDBMapper` parameters, see Optional configuration settings for DynamoDBMapper.

You can also set locking behavior for a specific operation only. For example, the following Java snippet uses the `DynamoDBMapper` to save a catalog item. It specifies `DynamoDBMapperConfig.SaveBehavior` by adding the optional `DynamoDBMapperConfig` parameter to the `save` method.

> **Note**
>
> The transactionWrite method does not support DynamoDBMapperConfig.SaveBehavior configuration. Disabling optimistic locking for transactionWrite is not supported.

Example

```
DynamoDBMapper mapper = new DynamoDBMapper(client);

// Load a catalog item.
CatalogItem item = mapper.load(CatalogItem.class, 101);
item.setTitle("This is a new title for the item");
...
```

```
// Save the item.
mapper.save(item,
    new DynamoDBMapperConfig(
        DynamoDBMapperConfig.SaveBehavior.CLOBBER));
```

Mapping arbitrary data in DynamoDB

In addition to the supported Java types (see Supported data types for DynamoDBMapper for Java), you can use types in your application for which there is no direct mapping to the Amazon DynamoDB types. To map these types, you must provide an implementation that converts your complex type to a DynamoDB supported type and vice versa, and annotate the complex type accessor method using the `@DynamoDBTypeConverted` annotation. The converter code transforms data when objects are saved or loaded. It is also used for all operations that consume complex types. Note that when comparing data during query and scan operations, the comparisons are made against the data stored in DynamoDB.

For example, consider the following `CatalogItem` class that defines a property, `Dimension`, that is of `DimensionType`. This property stores the item dimensions as height, width, and thickness. Assume that you decide to store these item dimensions as a string (such as 8.5x11x.05) in DynamoDB. The following example provides converter code that converts the `DimensionType` object to a string and a string to the `DimensionType`.

> **Note**
>
> This code example assumes that you have already loaded data into DynamoDB for your account by following the instructions in the Creating tables and loading data for code examples in DynamoDB section.
> For step-by-step instructions to run the following example, see Java code examples.

Example

```
public class DynamoDBMapperExample {

    static AmazonDynamoDB client;

    public static void main(String[] args) throws IOException {

        // Set the AWS region you want to access.
        Regions usWest2 = Regions.US_WEST_2;
```

```
        client = AmazonDynamoDBClientBuilder.standard().withRegion(usWest2).build();

        DimensionType dimType = new DimensionType();
        dimType.setHeight("8.00");
        dimType.setLength("11.0");
        dimType.setThickness("1.0");

        Book book = new Book();
        book.setId(502);
        book.setTitle("Book 502");
        book.setISBN("555-5555555555");
        book.setBookAuthors(new HashSet<String>(Arrays.asList("Author1", "Author2")));
        book.setDimensions(dimType);

        DynamoDBMapper mapper = new DynamoDBMapper(client);
        mapper.save(book);

        Book bookRetrieved = mapper.load(Book.class, 502);
        System.out.println("Book info: " + "\n" + bookRetrieved);

        bookRetrieved.getDimensions().setHeight("9.0");
        bookRetrieved.getDimensions().setLength("12.0");
        bookRetrieved.getDimensions().setThickness("2.0");

        mapper.save(bookRetrieved);

        bookRetrieved = mapper.load(Book.class, 502);
        System.out.println("Updated book info: " + "\n" + bookRetrieved);
    }

    @DynamoDBTable(tableName = "ProductCatalog")
    public static class Book {
        private int id;
        private String title;
        private String ISBN;
        private Set<String> bookAuthors;
        private DimensionType dimensionType;

        // Partition key
        @DynamoDBHashKey(attributeName = "Id")
        public int getId() {
            return id;
        }
```

```
        public void setId(int id) {
            this.id = id;
        }

        @DynamoDBAttribute(attributeName = "Title")
        public String getTitle() {
            return title;
        }

        public void setTitle(String title) {
            this.title = title;
        }

        @DynamoDBAttribute(attributeName = "ISBN")
        public String getISBN() {
            return ISBN;
        }

        public void setISBN(String ISBN) {
            this.ISBN = ISBN;
        }

        @DynamoDBAttribute(attributeName = "Authors")
        public Set<String> getBookAuthors() {
            return bookAuthors;
        }

        public void setBookAuthors(Set<String> bookAuthors) {
            this.bookAuthors = bookAuthors;
        }

        @DynamoDBTypeConverted(converter = DimensionTypeConverter.class)
        @DynamoDBAttribute(attributeName = "Dimensions")
        public DimensionType getDimensions() {
            return dimensionType;
        }

        @DynamoDBAttribute(attributeName = "Dimensions")
        public void setDimensions(DimensionType dimensionType) {
            this.dimensionType = dimensionType;
        }

        @Override
        public String toString() {
```

```
            return "Book [ISBN=" + ISBN + ", bookAuthors=" + bookAuthors + ",
dimensionType= "
                    + dimensionType.getHeight() + " X " + dimensionType.getLength() + "
X "
                    + dimensionType.getThickness()
                    + ", Id=" + id + ", Title=" + title + "]";
        }
    }

    static public class DimensionType {

        private String length;
        private String height;
        private String thickness;

        public String getLength() {
            return length;
        }

        public void setLength(String length) {
            this.length = length;
        }

        public String getHeight() {
            return height;
        }

        public void setHeight(String height) {
            this.height = height;
        }

        public String getThickness() {
            return thickness;
        }

        public void setThickness(String thickness) {
            this.thickness = thickness;
        }
    }

    // Converts the complex type DimensionType to a string and vice-versa.
    static public class DimensionTypeConverter implements DynamoDBTypeConverter<String,
DimensionType> {
```

```
        @Override
        public String convert(DimensionType object) {
            DimensionType itemDimensions = (DimensionType) object;
            String dimension = null;
            try {
                if (itemDimensions != null) {
                    dimension = String.format("%s x %s x %s",
 itemDimensions.getLength(), itemDimensions.getHeight(),
                            itemDimensions.getThickness());
                }
            } catch (Exception e) {
                e.printStackTrace();
            }
            return dimension;
        }

        @Override
        public DimensionType unconvert(String s) {

            DimensionType itemDimension = new DimensionType();
            try {
                if (s != null && s.length() != 0) {
                    String[] data = s.split("x");
                    itemDimension.setLength(data[0].trim());
                    itemDimension.setHeight(data[1].trim());
                    itemDimension.setThickness(data[2].trim());
                }
            } catch (Exception e) {
                e.printStackTrace();
            }

            return itemDimension;
        }
    }
}
```

DynamoDBMapper examples

The AWS SDK for Java provides a `DynamoDBMapper` class, allowing you to map your client-side classes to DynamoDB tables. To use `DynamoDBMapper`, you define the relationship between items in a DynamoDB table and their corresponding object instances in your code. The `DynamoDBMapper` class enables you to perform various create, read, update, and delete (CRUD) operations on items, and run queries and scans against tables.

To learn more about how to use `DynamoDBMapper`, see DynamoDB Examples Using the AWS SDK for Java in the *AWS SDK for Java 1.x Developer Guide*.

Java 2.x: DynamoDB Enhanced Client

The DynamoDB enhanced client is a high-level library that is part of the AWS SDK for Java version 2 (v2). It offers a straightforward way to map client-side classes to DynamoDB tables. You define the relationships between tables and their corresponding model classes in your code. After you define those relationships, you can intuitively perform various create, read, update, or delete (CRUD) operations on tables or items in DynamoDB.

For more information on how you can use the enhanced client with DynamoDB, see Using the DynamoDB Enhanced Client in the AWS SDK for Java 2.x .

Working with the .NET document model in DynamoDB

The AWS SDK for .NET provides document model classes that wrap some of the low-level Amazon DynamoDB operations, further simplifying your coding. In the document model, the primary classes are `Table` and `Document`. The `Table` class provides data operation methods such as `PutItem`, `GetItem`, and `DeleteItem`. It also provides the `Query` and the `Scan` methods. The `Document` class represents a single item in a table.

The preceding document model classes are available in the `Amazon.DynamoDBv2.DocumentModel` namespace.

> **Note**
>
> You can't use the document model classes to create, update, and delete tables. However, the document model does support most common data operations.

Topics

- Supported data types

Supported data types

The document model supports a set of primitive .NET data types and collections data types. The model supports the following primitive data types.

- `bool`
- `byte`
- `char`
- `DateTime`
- `decimal`
- `double`
- `float`
- `Guid`
- `Int16`
- `Int32`
- `Int64`
- `SByte`
- `string`
- `UInt16`
- `UInt32`
- `UInt64`

The following table summarizes the mapping of the preceding .NET types to the DynamoDB types.

.NET primitive type	DynamoDB type
All number types	`N` (number type)
All string types	`S` (string type)
MemoryStream, byte[]	`B` (binary type)
bool	`N` (number type). 0 represents false and 1 represents true.
DateTime	`S` (string type). The `DateTime` values are stored as ISO-8601 formatted strings.
Guid	`S` (string type).

.NET primitive type	DynamoDB type
Collection types (List, HashSet, and array)	BS (binary set) type, SS (string set) type, and NS (number set) type

AWS SDK for .NET defines types for mapping DynamoDB's Boolean, null, list and map types to .NET document model API:

- Use `DynamoDBBool` for Boolean type.
- Use `DynamoDBNull` for null type.
- Use `DynamoDBList` for list type.
- Use `Document` for map type.

Note

- Empty binary values are supported.
- Reading of empty string values is supported. Empty string attribute values are supported within attribute values of string Set type while writing to DynamoDB. Empty string attribute values of string type and empty string values contained within List or Map type are dropped from write requests

Working with the .NET object persistence model and DynamoDB

The AWS SDK for .NET provides an object persistence model that enables you to map your client-side classes to Amazon DynamoDB tables. Each object instance then maps to an item in the corresponding tables. To save your client-side objects to the tables, the object persistence model provides the `DynamoDBContext` class, an entry point to DynamoDB. This class provides you a connection to DynamoDB and enables you to access tables, perform various CRUD operations, and run queries.

The object persistence model provides a set of attributes to map client-side classes to tables, and properties/fields to table attributes.

> **Note**
>
> The object persistence model does not provide an API to create, update, or delete tables. It provides only data operations. You can use only the AWS SDK for .NET low-level API to create, update, and delete tables.

The following example shows how the object persistence model works. It starts with the `ProductCatalog` table. It has `Id` as the primary key.

```
ProductCatalog(Id, ...)
```

Suppose that you have a Book class with `Title`, `ISBN`, and `Authors` properties. You can map the Book class to the `ProductCatalog` table by adding the attributes defined by the object persistence model, as shown in the following C# code example.

Example

```
[DynamoDBTable("ProductCatalog")]
  public class Book
  {
    [DynamoDBHashKey]
    public int Id { get; set; }

    public string Title { get; set; }
    public int ISBN { get; set; }

    [DynamoDBProperty("Authors")]
    public List<string> BookAuthors { get; set; }

    [DynamoDBIgnore]
    public string CoverPage { get; set; }
  }
```

In the preceding example, the `DynamoDBTable` attribute maps the Book class to the `ProductCatalog` table.

The object persistence model supports both the explicit and default mapping between class properties and table attributes.

- **Explicit mapping—**To map a property to a primary key, you must use the `DynamoDBHashKey` and `DynamoDBRangeKey` object persistence model attributes. Additionally, for the nonprimary key attributes, if a property name in your class and the corresponding table attribute to which you want to map it are not the same, you must define the mapping by explicitly adding the `DynamoDBProperty` attribute.

  In the preceding example, the `Id` property maps to the primary key with the same name, and the `BookAuthors` property maps to the `Authors` attribute in the `ProductCatalog` table.

- **Default mapping—**By default, the object persistence model maps the class properties to the attributes with the same name in the table.

  In the preceding example, the properties `Title` and `ISBN` map to the attributes with the same name in the `ProductCatalog` table.

You don't have to map every single class property. You identify these properties by adding the `DynamoDBIgnore` attribute. When you save a `Book` instance to the table, the `DynamoDBContext` does not include the `CoverPage` property. It also does not return this property when you retrieve the book instance.

You can map properties of .NET primitive types such as int and string. You also can map any arbitrary data types as long as you provide an appropriate converter to map the arbitrary data to one of the DynamoDB types. To learn about mapping arbitrary types, see Mapping arbitrary data with DynamoDB using the AWS SDK for .NET object persistence model.

The object persistence model supports optimistic locking. During an update operation, this ensures that you have the latest copy of the item you are about to update. For more information, see Optimistic locking using DynamoDB and the AWS SDK for .NET object persistence model.

For more information, see the topics below.

Topics

- Supported data types
- DynamoDB attributes from the .NET object persistence model
- DynamoDBContext class from the .NET object persistence model
- Optimistic locking using DynamoDB and the AWS SDK for .NET object persistence model
- Mapping arbitrary data with DynamoDB using the AWS SDK for .NET object persistence model

Supported data types

The object persistence model supports a set of primitive .NET data types, collections, and arbitrary data types. The model supports the following primitive data types.

- `bool`
- `byte`
- `char`
- `DateTime`
- `decimal`
- `double`
- `float`
- `Int16`
- `Int32`
- `Int64`
- `SByte`
- `string`
- `UInt16`
- `UInt32`
- `UInt64`

The object persistence model also supports the .NET collection types. `DynamoDBContext` is able to convert concrete collection types and simple Plain Old CLR Objects (POCOs).

The following table summarizes the mapping of the preceding .NET types to the DynamoDB types.

.NET primitive type	DynamoDB type
All number types	`N` (number type)
All string types	`S` (string type)
MemoryStream, byte[]	`B` (binary type)

.NET primitive type	DynamoDB type
bool	N (number type). 0 represents false and 1 represents true.
Collection types	BS (binary set) type, SS (string set) type, and NS (number set) type
DateTime	S (string type). The `DateTime` values are stored as ISO-8601 formatted strings.

The object persistence model also supports arbitrary data types. However, you must provide converter code to map the complex types to the DynamoDB types.

Note

- Empty binary values are supported.
- Reading of empty string values is supported. Empty string attribute values are supported within attribute values of string Set type while writing to DynamoDB. Empty string attribute values of string type and empty string values contained within List or Map type are dropped from write requests

DynamoDB attributes from the .NET object persistence model

This section describes the attributes that the object persistence model offers so that you can map your classes and properties to DynamoDB tables and attributes.

Note

In the following attributes, only `DynamoDBTable` and `DynamoDBHashKey` are required.

DynamoDBGlobalSecondaryIndexHashKey

Maps a class property to the partition key of a global secondary index. Use this attribute if you need to `Query` a global secondary index.

DynamoDBGlobalSecondaryIndexRangeKey

Maps a class property to the sort key of a global secondary index. Use this attribute if you need to `Query` a global secondary index and want to refine your results using the index sort key.

DynamoDBHashKey

Maps a class property to the partition key of the table's primary key. The primary key attributes cannot be a collection type.

The following C# code example maps the `Book` class to the `ProductCatalog` table, and the `Id` property to the table's primary key partition key.

```
[DynamoDBTable("ProductCatalog")]
public class Book
{
    [DynamoDBHashKey]
    public int Id { get; set; }

    // Additional properties go here.
}
```

DynamoDBIgnore

Indicates that the associated property should be ignored. If you don't want to save any of your class properties, you can add this attribute to instruct `DynamoDBContext` not to include this property when saving objects to the table.

DynamoDBLocalSecondaryIndexRangeKey

Maps a class property to the sort key of a local secondary index. Use this attribute if you need to `Query` a local secondary index and want to refine your results using the index sort key.

DynamoDBProperty

Maps a class property to a table attribute. If the class property maps to a table attribute of the same name, you don't need to specify this attribute. However, if the names are not the same, you can use this tag to provide the mapping. In the following C# statement, the `DynamoDBProperty` maps the `BookAuthors` property to the `Authors` attribute in the table.

```
[DynamoDBProperty("Authors")]
```

```
public List<string> BookAuthors { get; set; }
```

DynamoDBContext uses this mapping information to create the Authors attribute when saving object data to the corresponding table.

DynamoDBRenamable

Specifies an alternative name for a class property. This is useful if you are writing a custom converter for mapping arbitrary data to a DynamoDB table where the name of a class property is different from a table attribute.

DynamoDBRangeKey

Maps a class property to the sort key of the table's primary key. If the table has a composite primary key (partition key and sort key), you must specify both the DynamoDBHashKey and DynamoDBRangeKey attributes in your class mapping.

For example, the sample table Reply has a primary key made of the Id partition key and Replenishment sort key. The following C# code example maps the Reply class to the Reply table. The class definition also indicates that two of its properties map to the primary key.

```
[DynamoDBTable("Reply")]
public class Reply
{
   [DynamoDBHashKey]
   public int ThreadId { get; set; }
   [DynamoDBRangeKey]
   public string Replenishment { get; set; }

   // Additional properties go here.
}
```

DynamoDBTable

Identifies the target table in DynamoDB to which the class maps. For example, the following C# code example maps the Developer class to the People table in DynamoDB.

```
[DynamoDBTable("People")]
public class Developer { ...}
```

This attribute can be inherited or overridden.

- The DynamoDBTable attribute can be inherited. In the preceding example, if you add a new class, Lead, that inherits from the Developer class, it also maps to the People table. Both the Developer and Lead objects are stored in the People table.
- The DynamoDBTable attribute can also be overridden. In the following C# code example, the Manager class inherits from the Developer class. However, the explicit addition of the DynamoDBTable attribute maps the class to another table (Managers).

```
[DynamoDBTable("Managers")]
public class Manager : Developer { ...}
```

You can add the optional parameter, LowerCamelCaseProperties, to request DynamoDB to make the first letter of the property name lowercase when storing the objects to a table, as shown in the following C# example.

```
[DynamoDBTable("People", LowerCamelCaseProperties=true)]
public class Developer
{
    string DeveloperName;
    ...
}
```

When saving instances of the Developer class, DynamoDBContext saves the DeveloperName property as the developerName.

DynamoDBVersion

Identifies a class property for storing the item version number. For more information about versioning, see Optimistic locking using DynamoDB and the AWS SDK for .NET object persistence model.

DynamoDBContext class from the .NET object persistence model

The DynamoDBContext class is the entry point to the Amazon DynamoDB database. It provides a connection to DynamoDB and enables you to access your data in various tables, perform various CRUD operations, and run queries. The DynamoDBContext class provides the following methods.

Topics

- CreateMultiTableBatchGet

- CreateMultiTableBatchWrite
- CreateBatchGet
- CreateBatchWrite
- Delete
- Dispose
- ExecuteBatchGet
- ExecuteBatchWrite
- FromDocument
- FromQuery
- FromScan
- GetTargetTable
- Load
- Query
- Save
- Scan
- ToDocument
- Specifying optional parameters for DynamoDBContext

CreateMultiTableBatchGet

Creates a `MultiTableBatchGet` object, composed of multiple individual `BatchGet` objects. Each of these `BatchGet` objects can be used for retrieving items from a single DynamoDB table.

To retrieve the items from tables, use the `ExecuteBatchGet` method, passing the `MultiTableBatchGet` object as a parameter.

CreateMultiTableBatchWrite

Creates a `MultiTableBatchWrite` object, composed of multiple individual `BatchWrite` objects. Each of these `BatchWrite` objects can be used for writing or deleting items in a single DynamoDB table.

To write to tables, use the `ExecuteBatchWrite` method, passing the `MultiTableBatchWrite` object as a parameter.

CreateBatchGet

Creates a `BatchGet` object that you can use to retrieve multiple items from a table.

CreateBatchWrite

Creates a `BatchWrite` object that you can use to put multiple items into a table, or to delete multiple items from a table.

Delete

Deletes an item from the table. The method requires the primary key of the item you want to delete. You can provide either the primary key value or a client-side object containing a primary key value as a parameter to this method.

- If you specify a client-side object as a parameter and you have enabled optimistic locking, the delete succeeds only if the client-side and the server-side versions of the object match.
- If you specify only the primary key value as a parameter, the delete succeeds regardless of whether you have enabled optimistic locking or not.

> **Note**
>
> To perform this operation in the background, use the `DeleteAsync` method instead.

Dispose

Disposes of all managed and unmanaged resources.

ExecuteBatchGet

Reads data from one or more tables, processing all of the `BatchGet` objects in a `MultiTableBatchGet`.

> **Note**
>
> To perform this operation in the background, use the `ExecuteBatchGetAsync` method instead.

ExecuteBatchWrite

Writes or deletes data in one or more tables, processing all of the `BatchWrite` objects in a `MultiTableBatchWrite`.

> **Note**
>
> To perform this operation in the background, use the `ExecuteBatchWriteAsync` method instead.

FromDocument

Given an instance of a `Document`, the `FromDocument` method returns an instance of a client-side class.

This is helpful if you want to use the document model classes along with the object persistence model to perform any data operations. For more information about the document model classes provided by the AWS SDK for .NET, see Working with the .NET document model in DynamoDB.

Suppose that you have a `Document` object named doc, that contains a representation of a `Forum` item. (To see how to construct this object, see the description for the `ToDocument` method later in this topic.) You can use `FromDocument` to retrieve the `Forum` item from the `Document`, as shown in the following C# code example.

Example

```
forum101 = context.FromDocument<Forum>(101);
```

> **Note**
>
> If your `Document` object implements the `IEnumerable` interface, you can use the `FromDocuments` method instead. This allows you to iterate over all of the class instances in the `Document`.

FromQuery

Runs a `Query` operation, with the query parameters defined in a `QueryOperationConfig` object.

> **Note**
>
> To perform this operation in the background, use the `FromQueryAsync` method instead.

FromScan

Runs a `Scan` operation, with the scan parameters defined in a `ScanOperationConfig` object.

> **Note**
>
> To perform this operation in the background, use the `FromScanAsync` method instead.

GetTargetTable

Retrieves the target table for the specified type. This is useful if you are writing a custom converter for mapping arbitrary data to a DynamoDB table, and you need to determine which table is associated with a custom data type.

Load

Retrieves an item from a table. The method requires only the primary key of the item you want to retrieve.

By default, DynamoDB returns the item with values that are eventually consistent. For information about the eventual consistency model, see DynamoDB read consistency.

`Load` or `LoadAsync` method calls the GetItem operation, which requires you to specify the primary key for the table. Because `GetItem` ignores the `IndexName` parameter, you can't load an item using an index's partition or sort key. Therefore, you must use the table's primary key to load an item.

> **Note**
>
> To perform this operation in the background, use the `LoadAsync` method instead. To view an example of using the `LoadAsync` method to perform high-level CRUD operations on a DynamoDB table, see the following example.

```
/// <summary>
/// Shows how to perform high-level CRUD operations on an Amazon DynamoDB
/// table.
/// </summary>
public class HighLevelItemCrud
{
    public static async Task Main()
    {
        var client = new AmazonDynamoDBClient();
        DynamoDBContext context = new DynamoDBContext(client);
        await PerformCRUDOperations(context);
    }

    public static async Task PerformCRUDOperations(IDynamoDBContext context)
    {
        int bookId = 1001; // Some unique value.
        Book myBook = new Book
        {
            Id = bookId,
            Title = "object persistence-AWS SDK for.NET SDK-Book 1001",
            Isbn = "111-1111111001",
            BookAuthors = new List<string> { "Author 1", "Author 2" },
        };

        // Save the book to the ProductCatalog table.
        await context.SaveAsync(myBook);

        // Retrieve the book from the ProductCatalog table.
        Book bookRetrieved = await context.LoadAsync<Book>(bookId);

        // Update some properties.
        bookRetrieved.Isbn = "222-2222221001";

        // Update existing authors list with the following values.
        bookRetrieved.BookAuthors = new List<string> { " Author 1", "Author x" };
        await context.SaveAsync(bookRetrieved);

        // Retrieve the updated book. This time, add the optional
        // ConsistentRead parameter using DynamoDBContextConfig object.
        await context.LoadAsync<Book>(bookId, new DynamoDBContextConfig
        {
            ConsistentRead = true,
```

```
            });

            // Delete the book.
            await context.DeleteAsync<Book>(bookId);

            // Try to retrieve deleted book. It should return null.
            Book deletedBook = await context.LoadAsync<Book>(bookId, new
 DynamoDBContextConfig
            {
                ConsistentRead = true,
            });

            if (deletedBook == null)
            {
                Console.WriteLine("Book is deleted");
            }
        }
    }
```

Query

Queries a table based on query parameters you provide.

You can query a table only if it has a composite primary key (partition key and sort key). When querying, you must specify a partition key and a condition that applies to the sort key.

Suppose that you have a client-side `Reply` class mapped to the `Reply` table in DynamoDB. The following C# code example queries the `Reply` table to find forum thread replies posted in the past 15 days. The `Reply` table has a primary key that has the `Id` partition key and the `ReplyDateTime` sort key.

Example

```
DynamoDBContext context = new DynamoDBContext(client);

string replyId = "DynamoDB#DynamoDB Thread 1"; //Partition key
DateTime twoWeeksAgoDate = DateTime.UtcNow.Subtract(new TimeSpan(14, 0, 0, 0)); // Date
 to compare.
IEnumerable<Reply> latestReplies = context.Query<Reply>(replyId,
 QueryOperator.GreaterThan, twoWeeksAgoDate);
```

This returns a collection of `Reply` objects.

The `Query` method returns a "lazy-loaded" `IEnumerable` collection. It initially returns only one page of results, and then makes a service call for the next page if needed. To obtain all the matching items, you need to iterate only over the `IEnumerable`.

If your table has a simple primary key (partition key), you can't use the `Query` method. Instead, you can use the `Load` method and provide the partition key to retrieve the item.

> **Note**
>
> To perform this operation in the background, use the `QueryAsync` method instead.

Save

Saves the specified object to the table. If the primary key specified in the input object doesn't exist in the table, the method adds a new item to the table. If the primary key exists, the method updates the existing item.

If you have optimistic locking configured, the update succeeds only if the client and the server-side versions of the item match. For more information, see Optimistic locking using DynamoDB and the AWS SDK for .NET object persistence model.

> **Note**
>
> To perform this operation in the background, use the `SaveAsync` method instead.

Scan

Performs an entire table scan.

You can filter scan results by specifying a scan condition. The condition can be evaluated on any attributes in the table. Suppose that you have a client-side class `Book` mapped to the `ProductCatalog` table in DynamoDB. The following C# example scans the table and returns only the book items priced less than 0.

Example

```
IEnumerable<Book> itemsWithWrongPrice = context.Scan<Book>(
                    new ScanCondition("Price", ScanOperator.LessThan, price),
```

```
                    new ScanCondition("ProductCategory", ScanOperator.Equal, "Book")
    );
```

The `Scan` method returns a "lazy-loaded" `IEnumerable` collection. It initially returns only one page of results, and then makes a service call for the next page if needed. To obtain all the matching items, you only need to iterate over the `IEnumerable`.

For performance reasons, you should query your tables and avoid a table scan.

> **Note**
>
> To perform this operation in the background, use the `ScanAsync` method instead.

ToDocument

Returns an instance of the `Document` document model class from your class instance.

This is helpful if you want to use the document model classes along with the object persistence model to perform any data operations. For more information about the document model classes provided by the AWS SDK for .NET, see Working with the .NET document model in DynamoDB.

Suppose that you have a client-side class mapped to the sample `Forum` table. You can then use a `DynamoDBContext` to get an item as a `Document` object from the `Forum` table, as shown in the following C# code example.

Example

```
DynamoDBContext context = new DynamoDBContext(client);

Forum forum101 = context.Load<Forum>(101); // Retrieve a forum by primary key.
Document doc = context.ToDocument<Forum>(forum101);
```

Specifying optional parameters for DynamoDBContext

When using the object persistence model, you can specify the following optional parameters for the `DynamoDBContext`.

- **`ConsistentRead`—**When retrieving data using the `Load`, `Query`, or `Scan` operations, you can add this optional parameter to request the latest values for the data.

- **`IgnoreNullValues`—**This parameter informs `DynamoDBContext` to ignore null values on attributes during a `Save` operation. If this parameter is false (or if it is not set), then a null value is interpreted as a directive to delete the specific attribute.
- **`SkipVersionCheck`—** This parameter informs `DynamoDBContext` not to compare versions when saving or deleting an item. For more information about versioning, see Optimistic locking using DynamoDB and the AWS SDK for .NET object persistence model.
- **`TableNamePrefix`—** Prefixes all table names with a specific string. If this parameter is null (or if it is not set), then no prefix is used.
- `DynamoDBEntryConversion` – Specifies the conversion schema that is used by the client. You can set this parameter to version V1 or V2. V1 is the default version.

  Based on the version that you set, the behavior of this parameter changes. For example:

  - In V1, the `bool` data type is converted to the `N` number type, where 0 represents false and 1 represents true. In V2, `bool` is converted to `BOOL`.
  - In V2, lists and arrays aren't grouped together with HashSets. Lists and arrays of numerics, string-based types, and binary-based types are converted to the `L` (List) type, which can be sent empty to update a list. This is unlike V1, where an empty list isn't sent over the wire.

    In V1, collection types, such as List, HashSet, and arrays are treated the same. List, HashSet, and array of numerics is converted to the `NS` (number set) type.

  The following example sets the conversion schema version to V2, which changes the conversion behavior between .NET types and DynamoDB data types.

```
var config = new DynamoDBContextConfig
{
    Conversion = DynamoDBEntryConversion.V2
};
var contextV2 = new DynamoDBContext(client, config);
```

The following C# example creates a new `DynamoDBContext` by specifying two of the preceding optional parameters, `ConsistentRead` and `SkipVersionCheck`.

Example

```
AmazonDynamoDBClient client = new AmazonDynamoDBClient();
...
```

```
DynamoDBContext context =
       new DynamoDBContext(client, new DynamoDBContextConfig { ConsistentRead = true,
 SkipVersionCheck = true});
```

`DynamoDBContext` includes these optional parameters with each request that you send using this context.

Instead of setting these parameters at the `DynamoDBContext` level, you can specify them for individual operations you run using `DynamoDBContext`, as shown in the following C# code example. The example loads a specific book item. The `Load` method of `DynamoDBContext` specifies the `ConsistentRead` and `SkipVersionCheck` optional parameters.

Example

```
AmazonDynamoDBClient client = new AmazonDynamoDBClient();
...
DynamoDBContext context = new DynamoDBContext(client);
Book bookItem = context.Load<Book>(productId,new DynamoDBContextConfig{ ConsistentRead
 = true, SkipVersionCheck = true });
```

In this case, `DynamoDBContext` includes these parameters only when sending the `Get` request.

Optimistic locking using DynamoDB and the AWS SDK for .NET object persistence model

Optimistic locking support in the object persistence model ensures that the item version for your application is the same as the item version on the server side before updating or deleting the item. Suppose that you retrieve an item for update. However, before you send your updates back, some other application updates the same item. Now your application has a stale copy of the item. Without optimistic locking, any update you perform will overwrite the update made by the other application.

The optimistic locking feature of the object persistence model provides the `DynamoDBVersion` tag that you can use to enable optimistic locking. To use this feature, you add a property to your class for storing the version number. You add the `DynamoDBVersion` attribute to the property. When you first save the object, the `DynamoDBContext` assigns a version number and increments this value each time you update the item.

Your update or delete request succeeds only if the client-side object version matches the corresponding version number of the item on the server side. If your application has a stale copy, it must get the latest version from the server before it can update or delete that item.

The following C# code example defines a Book class with object persistence attributes mapping it to the ProductCatalog table. The VersionNumber property in the class decorated with the DynamoDBVersion attribute stores the version number value.

Example

```
[DynamoDBTable("ProductCatalog")]
  public class Book
  {
    [DynamoDBHashKey]   //Partition key
    public int Id { get; set; }
    [DynamoDBProperty]
    public string Title { get; set; }
    [DynamoDBProperty]
    public string ISBN { get; set; }
    [DynamoDBProperty("Authors")]
    public List<string> BookAuthors { get; set; }
    [DynamoDBVersion]
    public int? VersionNumber { get; set; }
  }
```

Note

You can apply the DynamoDBVersion attribute only to a nullable numeric primitive type (such as int?).

Optimistic locking has the following impact on DynamoDBContext operations:

- For a new item, DynamoDBContext assigns initial version number 0. If you retrieve an existing item, update one or more of its properties, and try to save the changes, the save operation succeeds only if the version number on the client side and the server side match. DynamoDBContext increments the version number. You don't need to set the version number.
- The Delete method provides overloads that can take either a primary key value or an object as parameter, as shown in the following C# code example.

  Example

```
DynamoDBContext context = new DynamoDBContext(client);
...
```

```
// Load a book.
Book book = context.Load<ProductCatalog>(111);
// Do other operations.
// Delete 1 - Pass in the book object.
context.Delete<ProductCatalog>(book);

// Delete 2 - Pass in the Id (primary key)
context.Delete<ProductCatalog>(222);
```

If you provide an object as the parameter, the delete succeeds only if the object version matches the corresponding server-side item version. However, if you provide a primary key value as the parameter, `DynamoDBContext` is unaware of any version numbers, and it deletes the item without making the version check.

Note that the internal implementation of optimistic locking in the object persistence model code uses the conditional update and the conditional delete API actions in DynamoDB.

Disabling optimistic locking

To disable optimistic locking, you use the `SkipVersionCheck` configuration property. You can set this property when creating `DynamoDBContext`. In this case, optimistic locking is disabled for any requests that you make using the context. For more information, see Specifying optional parameters for DynamoDBContext.

Instead of setting the property at the context level, you can disable optimistic locking for a specific operation, as shown in the following C# code example. The example uses the context to delete a book item. The `Delete` method sets the optional `SkipVersionCheck` property to true, disabling version checking.

Example

```
DynamoDBContext context = new DynamoDBContext(client);
// Load a book.
Book book = context.Load<ProductCatalog>(111);
...
// Delete the book.
context.Delete<Book>(book, new DynamoDBContextConfig { SkipVersionCheck = true });
```

Mapping arbitrary data with DynamoDB using the AWS SDK for .NET object persistence model

In addition to the supported .NET types (see Supported data types), you can use types in your application for which there is no direct mapping to the Amazon DynamoDB types. The object persistence model supports storing data of arbitrary types as long as you provide the converter to convert data from the arbitrary type to the DynamoDB type and vice versa. The converter code transforms data during both the saving and loading of the objects.

You can create any types on the client-side. However the data stored in the tables is one of the DynamoDB types, and during query and scan, any data comparisons made are against the data stored in DynamoDB.

The following C# code example defines a `Book` class with `Id`, `Title`, `ISBN`, and `Dimension` properties. The `Dimension` property is of the `DimensionType` that describes `Height`, `Width`, and `Thickness` properties. The example code provides the converter methods `ToEntry` and `FromEntry` to convert data between the `DimensionType` and the DynamoDB string types. For example, when saving a `Book` instance, the converter creates a book `Dimension` string such as "8.5x11x.05". When you retrieve a book, it converts the string to a `DimensionType` instance.

The example maps the `Book` type to the `ProductCatalog` table. It saves a sample `Book` instance, retrieves it, updates its dimensions, and saves the updated `Book` again.

For step-by-step instructions for testing the following example, see .NET code examples.

Example

```
using System;
using System.Collections.Generic;
using Amazon.DynamoDBv2;
using Amazon.DynamoDBv2.DataModel;
using Amazon.DynamoDBv2.DocumentModel;
using Amazon.Runtime;
using Amazon.SecurityToken;

namespace com.amazonaws.codesamples
{
    class HighLevelMappingArbitraryData
    {
        private static AmazonDynamoDBClient client = new AmazonDynamoDBClient();

        static void Main(string[] args)
```

```
        {
            try
            {
                DynamoDBContext context = new DynamoDBContext(client);

                // 1. Create a book.
                DimensionType myBookDimensions = new DimensionType()
                {
                    Length = 8M,
                    Height = 11M,
                    Thickness = 0.5M
                };

                Book myBook = new Book
                {
                    Id = 501,
                    Title = "AWS SDK for .NET Object Persistence Model Handling
Arbitrary Data",
                    ISBN = "999-9999999999",
                    BookAuthors = new List<string> { "Author 1", "Author 2" },
                    Dimensions = myBookDimensions
                };

                context.Save(myBook);

                // 2. Retrieve the book.
                Book bookRetrieved = context.Load<Book>(501);

                // 3. Update property (book dimensions).
                bookRetrieved.Dimensions.Height += 1;
                bookRetrieved.Dimensions.Length += 1;
                bookRetrieved.Dimensions.Thickness += 0.2M;
                // Update the book.
                context.Save(bookRetrieved);

                Console.WriteLine("To continue, press Enter");
                Console.ReadLine();
            }
            catch (AmazonDynamoDBException e) { Console.WriteLine(e.Message); }
            catch (AmazonServiceException e) { Console.WriteLine(e.Message); }
            catch (Exception e) { Console.WriteLine(e.Message); }
        }
    }
    [DynamoDBTable("ProductCatalog")]
```

```
    public class Book
    {
        [DynamoDBHashKey] //Partition key
        public int Id
        {
            get; set;
        }
        [DynamoDBProperty]
        public string Title
        {
            get; set;
        }
        [DynamoDBProperty]
        public string ISBN
        {
            get; set;
        }
        // Multi-valued (set type) attribute.
        [DynamoDBProperty("Authors")]
        public List<string> BookAuthors
        {
            get; set;
        }
        // Arbitrary type, with a converter to map it to DynamoDB type.
        [DynamoDBProperty(typeof(DimensionTypeConverter))]
        public DimensionType Dimensions
        {
            get; set;
        }
    }

    public class DimensionType
    {
        public decimal Length
        {
            get; set;
        }
        public decimal Height
        {
            get; set;
        }
        public decimal Thickness
        {
            get; set;
```

```
        }
    }

    // Converts the complex type DimensionType to string and vice-versa.
    public class DimensionTypeConverter : IPropertyConverter
    {
        public DynamoDBEntry ToEntry(object value)
        {
            DimensionType bookDimensions = value as DimensionType;
            if (bookDimensions == null) throw new ArgumentOutOfRangeException();

            string data = string.Format("{1}{0}{2}{0}{3}", " x ",
                              bookDimensions.Length, bookDimensions.Height,
 bookDimensions.Thickness);

            DynamoDBEntry entry = new Primitive
            {
                Value = data
            };
            return entry;
        }

        public object FromEntry(DynamoDBEntry entry)
        {
            Primitive primitive = entry as Primitive;
            if (primitive == null || !(primitive.Value is String) ||
 string.IsNullOrEmpty((string)primitive.Value))
                throw new ArgumentOutOfRangeException();

            string[] data = ((string)(primitive.Value)).Split(new string[] { " x " },
 StringSplitOptions.None);
            if (data.Length != 3) throw new ArgumentOutOfRangeException();

            DimensionType complexData = new DimensionType
            {
                Length = Convert.ToDecimal(data[0]),
                Height = Convert.ToDecimal(data[1]),
                Thickness = Convert.ToDecimal(data[2])
            };
            return complexData;
        }
    }
}
```

Running the code examples in this Developer Guide

The AWS SDKs provide broad support for Amazon DynamoDB in the following languages:

- Java
- JavaScript in the browser
- .NET
- Node.js
- PHP
- Python
- Ruby
- C++
- Go
- Android
- iOS

The code examples in this developer guide provide more in-depth coverage of DynamoDB operations, using the following programming languages:

- Java code examples
- .NET code examples

Before you can begin with this exercise, you need to create an AWS account, get your access key and secret key, and set up the AWS Command Line Interface (AWS CLI) on your computer. For more information, see Setting up DynamoDB (web service).

> **Note**
>
> If you are using the downloadable version of DynamoDB, you need to use the AWS CLI to create the tables and sample data. You also need to specify the `--endpoint-url` parameter with each AWS CLI command. For more information, see Setting the local endpoint.

Creating tables and loading data for code examples in DynamoDB

See below for the basics on creating tables in DynamoDB, loading in a sample dataset, querying the data, and updating the data.

- Step 1: Create a table in DynamoDB
- Step 2: Write data to a DynamoDB table
- Step 3: Read data from a DynamoDB table
- Step 4: Update data in a DynamoDB table

Java code examples

Topics

- Java: Setting your AWS credentials
- Java: Setting the AWS Region and endpoint

This Developer Guide contains Java code snippets and ready-to-run programs. You can find these code examples in the following sections:

- Working with items and attributes in DynamoDB
- Working with tables and data in DynamoDB
- Querying tables in DynamoDB
- Scanning tables in DynamoDB
- Improving data access with secondary indexes in DynamoDB
- Java 1.x: DynamoDBMapper
- Change data capture for DynamoDB Streams

You can get started quickly by using Eclipse with the AWS Toolkit for Eclipse. In addition to a full-featured IDE, you also get the AWS SDK for Java with automatic updates, and preconfigured templates for building AWS applications.

To run the Java code examples (using Eclipse)

1. Download and install the Eclipse IDE.

2. Download and install the AWS Toolkit for Eclipse.
3. Start Eclipse, and on the **Eclipse** menu, choose **File**, **New**, and then **Other**.
4. In **Select a wizard**, choose **AWS**, choose **AWS Java Project**, and then choose **Next**.
5. In **Create an AWS Java**, do the following:

   a. In **Project name**, enter a name for your project.

   b. In **Select Account**, choose your credentials profile from the list.

      If this is your first time using the AWS Toolkit for Eclipse, choose **Configure AWS Accounts** to set up your AWS credentials.
6. Choose **Finish** to create the project.
7. From the **Eclipse** menu, choose **File**, **New**, and then **Class**.
8. In **Java Class**, enter a name for your class in **Name** (use the same name as the code example that you want to run), and then choose **Finish** to create the class.
9. Copy the code example from the documentation page into the Eclipse editor.
10. To run the code, choose **Run** on the Eclipse menu.

The SDK for Java provides thread-safe clients for working with DynamoDB. As a best practice, your applications should create one client and reuse the client between threads.

For more information, see the AWS SDK for Java.

> **Note**
>
> The code examples in this guide are intended for use with the latest version of the AWS SDK for Java.
> If you are using the AWS Toolkit for Eclipse, you can configure automatic updates for the SDK for Java. To do this in Eclipse, go to **Preferences** and choose **AWS Toolkit**, **AWS SDK for Java**, **Download new SDKs automatically**.

Java: Setting your AWS credentials

The SDK for Java requires that you provide AWS credentials to your application at runtime. The code examples in this guide assume that you are using an AWS credentials file, as described in Set up your AWS credentials in the *AWS SDK for Java Developer Guide*.

The following is an example of an AWS credentials file named `~/.aws/credentials`, where the tilde character (~) represents your home directory.

```
[default]
aws_access_key_id = AWS access key ID goes here
aws_secret_access_key = Secret key goes here
```

Java: Setting the AWS Region and endpoint

By default, the code examples access DynamoDB in the US West (Oregon) Region. You can change the Region by modifying the `AmazonDynamoDB` properties.

The following code example instantiates a new `AmazonDynamoDB`.

```
import software.amazon.dynamodb.AmazonDynamoDBClientBuilder;
import com.amazonaws.regions.Regions;
...
// This client will default to US West (Oregon)
AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard()
.withRegion(Regions.US_WEST_2)
.build();
```

You can use the `withRegion` method to run your code against DynamoDB in any Region where it is available. For a complete list, see AWS regions and endpoints in the *Amazon Web Services General Reference*.

If you want to run the code examples using DynamoDB locally on your computer, set the endpoint as follows.

AWS SDK V1

```
AmazonDynamoDB client =
 AmazonDynamoDBClientBuilder.standard().withEndpointConfiguration(
new AwsClientBuilder.EndpointConfiguration("http://localhost:8000", "us-west-2"))
.build();
```

AWS SDK V2

```
DynamoDbClient client = DynamoDbClient.builder()
    .endpointOverride(URI.create("http://localhost:8000"))
```

```
    // The region is meaningless for local DynamoDb but required for client builder
validation
    .region(Region.US_EAST_1)
    .credentialsProvider(StaticCredentialsProvider.create(
    AwsBasicCredentials.create("dummy-key", "dummy-secret")))
    .build();
```

.NET code examples

Topics

- .NET: Setting your AWS credentials
- .NET: Setting the AWS Region and endpoint

This guide contains .NET code snippets and ready-to-run programs. You can find these code examples in the following sections:

- Working with items and attributes in DynamoDB
- Working with tables and data in DynamoDB
- Querying tables in DynamoDB
- Scanning tables in DynamoDB
- Improving data access with secondary indexes in DynamoDB
- Working with the .NET document model in DynamoDB
- Working with the .NET object persistence model and DynamoDB
- Change data capture for DynamoDB Streams

You can get started quickly by using the AWS SDK for .NET with the Toolkit for Visual Studio.

To run the .NET code examples (using Visual Studio)

1. Download and install Microsoft Visual Studio.
2. (Optional) Download and install the Toolkit for Visual Studio.
3. Set up your AWS credentials. Configure a credentials profile in your shared AWS credentials file (`~/.aws/credentials`). For more information, see Configure AWS credentials in the *AWS SDK for .NET Developer Guide*.
4. Start Visual Studio. Choose **File**, **New**, **Project**.

5. Search for **Console App**, select the C# template that targets .NET, and then choose **Next**. Configure your project name and location, and then choose **Create**.
6. Add the AWS SDK for DynamoDB NuGet package to your project:
   a. In Solution Explorer, open the context (right-click) menu for your project, and then choose **Manage NuGet Packages**.
   b. In NuGet Package Manager, choose **Browse**.
   c. In the search box, enter **`AWSSDK.DynamoDBv2`**, and wait for the search to complete.
   d. Choose **AWSSDK.DynamoDBv2**, and then choose **Install**.
7. In your Visual Studio project, open `Program.cs`. Replace the contents with the code example from the documentation page that you want to run.
8. To run the code, choose **Start** in the Visual Studio toolbar.

The SDK for .NET provides thread-safe clients for working with DynamoDB. As a best practice, your applications should create one client and reuse the client between threads.

For more information, see AWS SDK for .NET.

> **Note**
>
> The code examples in this guide are intended for use with the latest version of the AWS SDK for .NET.

.NET: Setting your AWS credentials

The SDK for .NET requires that you provide AWS credentials to your application at runtime. The code examples in this guide assume that you are using the SDK Store to manage your AWS credentials file, as described in Using the SDK store in the *AWS SDK for .NET Developer Guide*.

The Toolkit for Visual Studio supports multiple sets of credentials from any number of accounts. Each set is referred to as a *profile*. Visual Studio adds entries to the project's `App.config` file so that your application can find the AWS credentials at runtime.

The following example shows the default `App.config` file that is generated when you create a new project using Toolkit for Visual Studio.

```
<?xml version="1.0" encoding="utf-8" ?>
```

```
<configuration>
    <appSettings>
    <add key="AWSProfileName" value="default"/>
    <add key="AWSRegion" value="us-west-2" />
 </appSettings>
</configuration>
```

At runtime, the program uses the `default` set of AWS credentials, as specified by the `AWSProfileName` entry. The AWS credentials themselves are kept in the SDK Store in encrypted form. The Toolkit for Visual Studio provides a graphical user interface for managing your credentials, all from within Visual Studio. For more information, see Specifying credentials in the *AWS Toolkit for Visual Studio User Guide*.

> **Note**
>
> By default, the code examples access DynamoDB in the US West (Oregon) Region. You can change the Region by modifying the `AWSRegion` entry in the App.config file. You can set `AWSRegion` to any Region where DynamoDB is available. For a complete list, see AWS regions and endpoints in the *Amazon Web Services General Reference*.

.NET: Setting the AWS Region and endpoint

By default, the code examples access DynamoDB in the US West (Oregon) Region. You can change the Region by modifying the `AWSRegion` entry in the `App.config` file. Or, you can change the Region by modifying the `AmazonDynamoDBClient` properties.

The following code example instantiates a new `AmazonDynamoDBClient`. The client is modified so that the code runs against DynamoDB in a different Region.

```
AmazonDynamoDBConfig clientConfig = new AmazonDynamoDBConfig();
// This client will access the US East 1 region.
clientConfig.RegionEndpoint = RegionEndpoint.USEast1;
AmazonDynamoDBClient client = new AmazonDynamoDBClient(clientConfig);
```

For a complete list of Regions, see AWS regions and endpoints in the *Amazon Web Services General Reference*.

If you want to run the code examples using DynamoDB locally on your computer, set the endpoint as follows.

```
AmazonDynamoDBConfig clientConfig = new AmazonDynamoDBConfig();
// Set the endpoint URL
clientConfig.ServiceURL = "http://localhost:8000";
AmazonDynamoDBClient client = new AmazonDynamoDBClient(clientConfig);
```

DynamoDB low-level API

The Amazon DynamoDB *low-level API* is the protocol-level interface for DynamoDB. At this level, every HTTP(S) request must be correctly formatted and carry a valid digital signature.

The AWS SDKs construct low-level DynamoDB API requests on your behalf and process the responses from DynamoDB. This lets you focus on your application logic, instead of low-level details. However, you can still benefit from a basic knowledge of how the low-level DynamoDB API works.

For more information about the low-level DynamoDB API, see Amazon DynamoDB API Reference.

> **Note**
>
> DynamoDB Streams has its own low-level API, which is separate from that of DynamoDB and is fully supported by the AWS SDKs.
> For more information, see Change data capture for DynamoDB Streams. For the low-level DynamoDB Streams API, see the Amazon DynamoDB Streams API Reference.

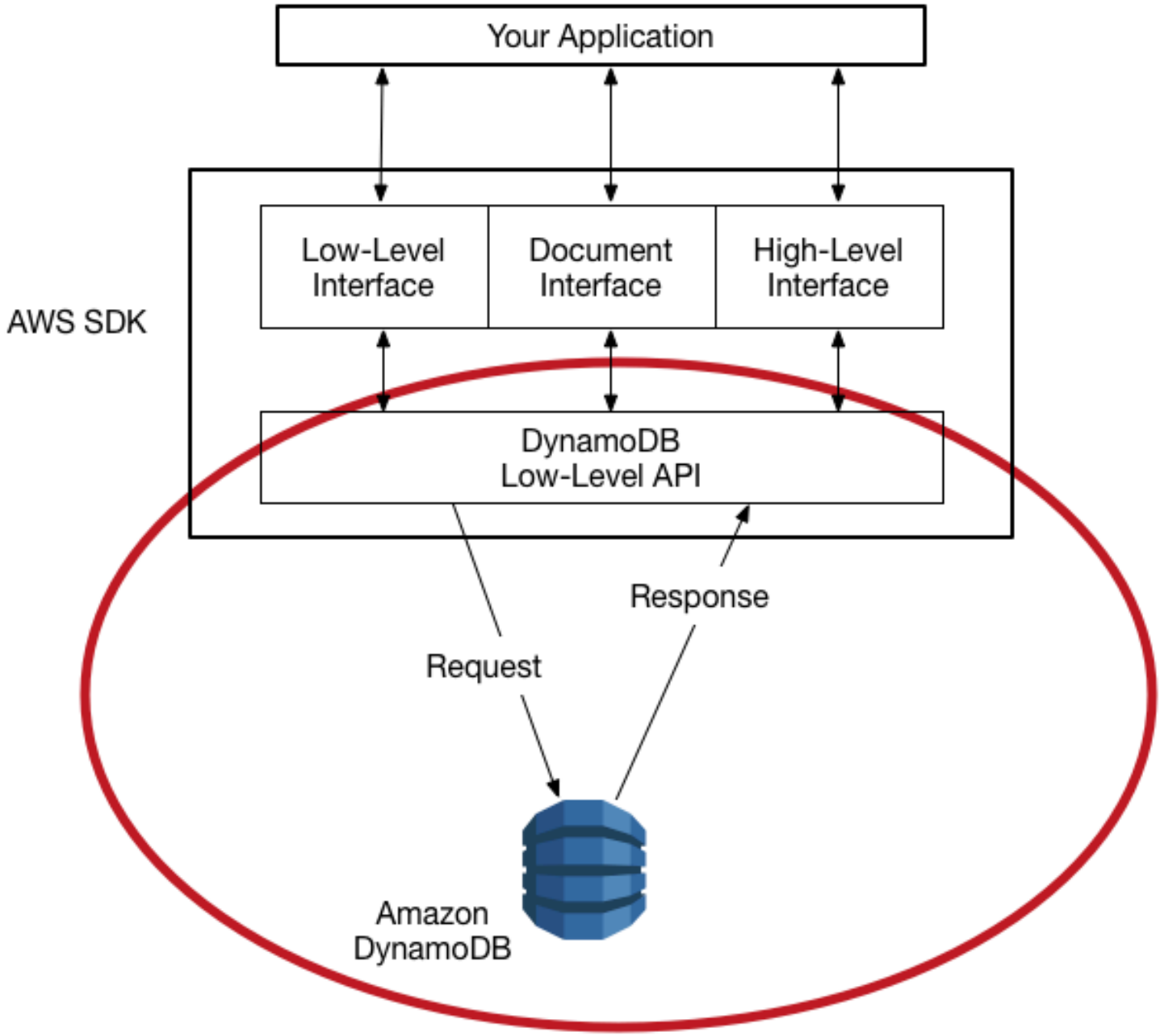
The low-level DynamoDB API uses JavaScript Object Notation (JSON) as a wire protocol format. JSON presents data in a hierarchy so that both data values and data structure are conveyed simultaneously. Name-value pairs are defined in the format `name:value`. The data hierarchy is defined by nested brackets of name-value pairs.

DynamoDB uses JSON only as a transport protocol, not as a storage format. The AWS SDKs use JSON to send data to DynamoDB, and DynamoDB responds with JSON. DynamoDB does not store data persistently in JSON format.

> **Note**
>
> For more information about JSON, see Introducing JSON on the `JSON.org` website.

Topics

- Request format
- Response format
- Data type descriptors
- Numeric data
- Binary data



Request format

The DynamoDB low-level API accepts HTTP(S) POST requests as input. The AWS SDKs construct these requests for you.

Suppose that you have a table named `Pets`, with a key schema consisting of `AnimalType` (partition key) and `Name` (sort key). Both of these attributes are of type `string`. To retrieve an item from `Pets`, the AWS SDK constructs the following request.

```
POST / HTTP/1.1
Host: dynamodb.<region>.<domain>;
Accept-Encoding: identity
Content-Length: <PayloadSizeBytes>
User-Agent: <UserAgentString>
Content-Type: application/x-amz-json-1.0
Authorization: AWS4-HMAC-SHA256 Credential=<Credential>, SignedHeaders=<Headers>,
 Signature=<Signature>
X-Amz-Date: <Date>
X-Amz-Target: DynamoDB_20120810.GetItem

{
    "TableName": "Pets",
    "Key": {
        "AnimalType": {"S": "Dog"},
        "Name": {"S": "Fido"}
    }
}
```

Note the following about this request:

- The `Authorization` header contains information required for DynamoDB to authenticate the request. For more information, see Signing AWS API requests and Signature Version 4 signing process in the *Amazon Web Services General Reference*.
- The `X-Amz-Target` header contains the name of a DynamoDB operation: `GetItem`. (This is also accompanied by the low-level API version, in this case `20120810`.)
- The payload (body) of the request contains the parameters for the operation, in JSON format. For the `GetItem` operation, the parameters are `TableName` and `Key`.

Response format

Upon receipt of the request, DynamoDB processes it and returns a response. For the request shown previously, the HTTP(S) response payload contains the results from the operation, as shown in the following example.

```
HTTP/1.1 200 OK
x-amzn-RequestId: <RequestId>
x-amz-crc32: <Checksum>
Content-Type: application/x-amz-json-1.0
Content-Length: <PayloadSizeBytes>
Date: <Date>
{
    "Item": {
        "Age": {"N": "8"},
        "Colors": {
            "L": [
                {"S": "White"},
                {"S": "Brown"},
                {"S": "Black"}
            ]
        },
        "Name": {"S": "Fido"},
        "Vaccinations": {
            "M": {
                "Rabies": {
                    "L": [
                        {"S": "2009-03-17"},
                        {"S": "2011-09-21"},
                        {"S": "2014-07-08"}
                    ]
                },
                "Distemper": {"S": "2015-10-13"}
            }
        },
        "Breed": {"S": "Beagle"},
        "AnimalType": {"S": "Dog"}
    }
}
```

At this point, the AWS SDK returns the response data to your application for further processing.

> **Note**
>
> If DynamoDB can't process a request, it returns an HTTP error code and message. The AWS SDK propagates these to your application in the form of exceptions. For more information, see Error handling with DynamoDB.

Data type descriptors

The low-level DynamoDB API protocol requires each attribute to be accompanied by a data type descriptor. *Data type descriptors* are tokens that tell DynamoDB how to interpret each attribute.

The examples in Request format and Response format show examples of how data type descriptors are used. The `GetItem` request specifies `S` for the `Pets` key schema attributes (`AnimalType` and `Name`), which are of type `string`. The `GetItem` response contains a *Pets* item with attributes of type `string` (`S`), `number` (`N`), `map` (`M`), and `list` (`L`).

The following is a complete list of DynamoDB data type descriptors:

- **`S`** – String
- **`N`** – Number
- **`B`** – Binary
- **`BOOL`** – Boolean
- **`NULL`** – Null
- **`M`** – Map
- **`L`** – List
- **`SS`** – String Set
- **`NS`** – Number Set
- **`BS`** – Binary Set

The following table shows the correct JSON format for each data type descriptor. Note that numbers are represented as strings to preserve precision, while booleans and nulls use their native JSON types.

Descriptor	JSON format	Notes
S	`{"S": "Hello"}`	Value is a JSON string.
N	`{"N": "123.45"}`	Value is a string, not a JSON number. This preserves precision across languages.
B	`{"B": "dGhpcyBpcyBhIHRlc3Q="}`	Value is a base64-encoded string.
BOOL	`{"BOOL": true}`	Value is a JSON boolean (`true` or `false`), not a string.
NULL	`{"NULL": true}`	Value is the JSON boolean `true` to indicate null.
M	`{"M": {"Name": {"S": "Joe"}}}`	Value is a JSON object of attribute name–value pairs.
L	`{"L": [{"S": "Red"}, {"N": "5"}]}`	Value is a JSON array of attribute values.
SS	`{"SS": ["Red", "Blue"]}`	Value is a JSON array of strings.
NS	`{"NS": ["1", "2.5"]}`	Value is a JSON array of number strings.
BS	`{"BS": ["U3Vubnk=", "UmFpbnk="]}`	Value is a JSON array of base64-encoded strings.

Note

For detailed descriptions of DynamoDB data types, see Data types.

Numeric data

Different programming languages offer different levels of support for JSON. In some cases, you might decide to use a third-party library for validating and parsing JSON documents.

Some third-party libraries build upon the JSON number type, providing their own types such as `int`, `long`, or `double`. However, the native number data type in DynamoDB does not map exactly to these other data types, so these type distinctions can cause conflicts. In addition, many JSON libraries do not handle fixed-precision numeric values, and they automatically infer a double data type for digit sequences that contain a decimal point.

To solve these problems, DynamoDB provides a single numeric type with no data loss. To avoid unwanted implicit conversions to a double value, DynamoDB uses strings for the data transfer of numeric values. This approach provides flexibility for updating attribute values while maintaining proper sorting semantics, such as putting the values "01", "2", and "03" in the proper sequence.

If number precision is important to your application, you should convert numeric values to strings before you pass them to DynamoDB.

Binary data

DynamoDB supports binary attributes. However, JSON does not natively support encoding binary data. To send binary data in a request, you will need to encode it in base64 format. Upon receiving the request, DynamoDB decodes the base64 data back to binary.

The base64 encoding scheme used by DynamoDB is described at RFC 4648 on the Internet Engineering Task Force (IETF) website.

Programming Amazon DynamoDB with Python and Boto3

This guide provides an orientation to programmers wanting to use Amazon DynamoDB with Python. Learn about the different abstraction layers, configuration management, error handling, controlling retry policies, managing keep-alive, and more.

Topics

- About Boto
- Using the Boto documentation
- Understanding the client and resource abstraction layers
- Using the table resource batch_writer

- Additional code examples that explore the client and resource layers
- Understanding how the Client and Resource objects interact with sessions and threads
- Customizing the Config object
- Error handling
- Logging
- Event hooks
- Pagination and the Paginator
- Waiters

About Boto

You can access DynamoDB from Python by using the official AWS SDK for Python, commonly referred to as **Boto3**. The name Boto (pronounced boh-toh) comes from a freshwater dolphin native to the Amazon River. The Boto3 library is the library's third major version, first released in 2015. The Boto3 library is quite large, as it supports all AWS services, not just DynamoDB. This orientation targets only the parts of Boto3 relevant to DynamoDB.

Boto is maintained and published by AWS as open-source project hosted on GitHub. It's split into two packages: Botocore and Boto3.

- **Botocore** provides the low-level functionality. In Botocore you'll find the client, session, credentials, config, and exception classes.
- **Boto3** builds on top of Botocore. It offers a higher-level, more Pythonic interface. Specifically, it exposes a DynamoDB table as a Resource and offers a simpler, more elegant interface compared to the lower-level, service-oriented client interface.

Because these projects are hosted on GitHub, you can view the source code, track open issues, or submit your own issues.

Using the Boto documentation

Get started with the Boto documentation with the following resources:

- Begin with the Quickstart section that provides a solid starting point for the package installation. Go there for instructions on getting Boto3 installed if it's not already (Boto3 is often automatically available within AWS services such as AWS Lambda).

- After that, focus on the documentation's DynamoDB guide. It shows you how to perform the basic DynamoDB activities: create and delete a table, manipulate items, run batch operations, run a query, and perform a scan. Its examples use the **resource** interface. When you see `boto3.resource('dynamodb')` that indicates you're using the higher-level **resource** interface.
- After the guide, you can review the DynamoDB reference. This landing page provides an exhaustive list of the classes and methods available to you. At the top, you'll see the `DynamoDB.Client` class. This provides low-level access to all the control-plane and data-plane operations. At the bottom, look at the `DynamoDB.ServiceResource` class. This is the higher-level Pythonic interface. With it you can create a table, do batch operations across tables, or obtain a `DynamoDB.ServiceResource.Table` instance for table-specific actions.

Understanding the client and resource abstraction layers

The two interfaces you'll be working with are the **client** interface and the **resource** interface.

- The low-level **client** interface provides a 1-to-1 mapping to the underlying service API. Every API offered by DynamoDB is available through the client. This means the client interface can provide complete functionality, but it's often more verbose and complex to use.
- The higher-level **resource** interface does not provide a 1-to-1 mapping of the underlying service API. However, it provides methods that make it more convenient for you to access the service such as `batch_writer`.

Here's an example of inserting an item using the client interface. Notice how all values are passed as a map with the key indicating their type ('S' for string, 'N' for number) and their value as a string. This is known as DynamoDB JSON format.

```
import boto3

dynamodb = boto3.client('dynamodb')

dynamodb.put_item(
    TableName='YourTableName',
    Item={
        'pk': {'S': 'id#1'},
        'sk': {'S': 'cart#123'},
        'name': {'S': 'SomeName'},
        'inventory': {'N': '500'},
```

```
        # ... more attributes ...
    }
)
```

Here's the same `PutItem` operation using the resource interface. The data typing is implicit:

```
import boto3

dynamodb = boto3.resource('dynamodb')

table = dynamodb.Table('YourTableName')

table.put_item(
    Item={
        'pk': 'id#1',
        'sk': 'cart#123',
        'name': 'SomeName',
        'inventory': 500,
        # ... more attributes ...
    }
)
```

If needed, you can convert between regular JSON and DynamoDB JSON using the `TypeSerializer` and `TypeDeserializer` classes provided with boto3:

```
def dynamo_to_python(dynamo_object: dict) -> dict:
    deserializer = TypeDeserializer()
    return {
        k: deserializer.deserialize(v)
        for k, v in dynamo_object.items()
    }

def python_to_dynamo(python_object: dict) -> dict:
    serializer = TypeSerializer()
    return {
        k: serializer.serialize(v)
        for k, v in python_object.items()
    }
```

Here is how to perform a query using the client interface. It expresses the query as a JSON construct. It uses a `KeyConditionExpression` string which requires variable substitution to handle any potential keyword conflicts:

```
import boto3

client = boto3.client('dynamodb')

# Construct the query
response = client.query(
    TableName='YourTableName',
    KeyConditionExpression='pk = :pk_val AND begins_with(sk, :sk_val)',
    FilterExpression='#name = :name_val',
    ExpressionAttributeValues={
        ':pk_val': {'S': 'id#1'},
        ':sk_val': {'S': 'cart#'},
        ':name_val': {'S': 'SomeName'},
    },
    ExpressionAttributeNames={
        '#name': 'name',
    }
)
```

The same query operation using the resource interface can be shortened and simplified:

```
import boto3
from boto3.dynamodb.conditions import Key, Attr

dynamodb = boto3.resource('dynamodb')
table = dynamodb.Table('YourTableName')

response = table.query(
    KeyConditionExpression=Key('pk').eq('id#1') & Key('sk').begins_with('cart#'),
    FilterExpression=Attr('name').eq('SomeName')
)
```

As a final example, imagine you want to get the approximate size of a table (which is metadata kept on the table that is updated about every 6 hours). With the client interface, you do a `describe_table()` operation and pull the answer from the JSON structure returned:

```
import boto3
```

```
dynamodb = boto3.client('dynamodb')

response = dynamodb.describe_table(TableName='YourTableName')
size = response['Table']['TableSizeBytes']
```

With the resource interface, the table performs the describe operation implicitly and presents the data directly as an attribute:

```
import boto3

dynamodb = boto3.resource('dynamodb')

table = dynamodb.Table('YourTableName')
size = table.table_size_bytes
```

> **Note**
>
> When considering whether to develop using the client or resource interface, be aware that new features will not be added to the resource interface per the resource documentation: "The AWS Python SDK team does not intend to add new features to the resources interface in boto3. Existing interfaces will continue to operate during boto3's lifecycle. Customers can find access to newer service features through the client interface."

Using the table resource batch_writer

One convenience available only with the higher-level table resource is the `batch_writer`. DynamoDB supports batch write operations allowing up to 25 put or delete operations in one network request. Batching like this improves efficiency by minimizing network round trips.

With the low-level client library, you use the `client.batch_write_item()` operation to run batches. You must manually split your work into batches of 25. After each operation, you also have to request to receive a list of unprocessed items (some of the write operations may succeed while others could fail). You then have to pass those unprocessed items again into a later `batch_write_item()` operation. There's a significant amount of boilerplate code.

The Table.batch_writer method creates a context manager for writing objects in a batch. It presents an interface where it seems as if you're writing items one at a time, but internally it's buffering and sending the items in batches. It also handles unprocessed item retries implicitly.

```
dynamodb = boto3.resource('dynamodb')

table = dynamodb.Table('YourTableName')

movies = # long list of movies in {'pk': 'val', 'sk': 'val', etc} format
with table.batch_writer() as writer:
    for movie in movies:
        writer.put_item(Item=movie)
```

Additional code examples that explore the client and resource layers

You can also refer to the following code sample repositories that explore usage of the various functions, using both client and resource:

- Official AWS single-action code examples.
- Official AWS scenario-oriented code examples.
- Community-maintained single-action code examples.

Understanding how the Client and Resource objects interact with sessions and threads

The Resource object is not thread safe and should not be shared across threads or processes. Refer to the guide on Resource for more details.

The Client object, in contrast, is generally thread safe, except for specific advanced features. Refer to the guide on Clients for more details.

The Session object is not thread safe. So, each time you make a Client or Resource in a multi-threaded environment you should create a new Session first and then make the Client or Resource from the Session. Refer to the guide on Sessions for more details.

When you call the `boto3.resource()`, you're implicitly using the default Session. This is convenient for writing single-threaded code. When writing multi-threaded code, you'll want to first construct a new Session for each thread and then retrieve the resource from that Session:

```
# Explicitly create a new Session for this thread
session = boto3.Session()
dynamodb = session.resource('dynamodb')
```

Customizing the Config object

When constructing a Client or Resource object, you can pass optional named parameters to customize behavior. The parameter named `config` unlocks a variety of functionality. It's an instance of `botocore.client.Config` and the reference documentation for Config shows everything it exposes for you to control. The guide to Configuration provides a good overview.

> **Note**
>
> You can modify many of these behavioral settings at the Session level, within the AWS configuration file, or as environment variables.

Config for timeouts

One use of a custom config is to adjust networking behaviors:

- **connect_timeout (float or int)** – The time in seconds till a timeout exception is thrown when attempting to make a connection. The default is 60 seconds.
- **read_timeout (float or int)** – The time in seconds till a timeout exception is thrown when attempting to read from a connection. The default is 60 seconds.

Timeouts of 60 seconds are excessive for DynamoDB. It means a transient network glitch will cause a minute's delay for the client before it can try again. The following code shortens the timeouts to a second:

```
import boto3
from botocore.config import Config

my_config = Config(
   connect_timeout = 1.0,
   read_timeout = 1.0
)
dynamodb = boto3.resource('dynamodb', config=my_config)
```

For more discussion about timeouts, see Tuning AWS Java SDK HTTP request settings for latency-aware DynamoDB applications. Note the Java SDK has more timeout configurations than Python.

Config for keep-alive

If you're using botocore 1.27.84 or later, you can also control **TCP Keep-Alive**:

- **tcp_keepalive** (bool) - Enables the TCP Keep-Alive socket option used when creating new connections if set to `True` (defaults to `False`). This is only available starting with botocore 1.27.84.

Setting TCP Keep-Alive to `True` can reduce average latencies. Here's sample code that conditionally sets TCP Keep-Alive to true when you have the right botocore version:

```
import botocore
import boto3
from botocore.config import Config
from distutils.version import LooseVersion

required_version = "1.27.84"
current_version = botocore.__version__

my_config = Config(
   connect_timeout = 0.5,
   read_timeout = 0.5
)
if LooseVersion(current_version) > LooseVersion(required_version):
    my_config = my_config.merge(Config(tcp_keepalive = True))

dynamodb = boto3.resource('dynamodb', config=my_config)
```

Note

TCP Keep-Alive is different than HTTP Keep-Alive. With TCP Keep-Alive, small packets are sent by the underlying operating system over the socket connection to keep the connection alive and immediately detect any drops. With HTTP Keep-Alive, the web connection built on the underlying socket gets reused. HTTP Keep-Alive is always enabled with boto3.

There's a limit to how long an idle connection can be kept alive. Consider sending periodic requests (say every minute) if you have an idle connection but want the next request to use an already-established connection.

Config for retries

The config also accepts a dictionary called **retries** where you can specify your desired retry behavior. Retries happen within the SDK when the SDK receives an error and the error is of a transient type. If an error is retried internally (and a retry eventually produces a successful response), there's no error seen from the calling code's perspective, just a slightly elevated latency. Here are the values you can specify:

- **max_attempts** – An integer representing the maximum number of retry attempts that will be made on a single request. For example, setting this value to 2 will result in the request being retried at most two times after the initial request. Setting this value to 0 will result in no retries ever being attempted after the initial request.
- **total_max_attempts** – An integer representing the maximum number of total attempts that will be made on a single request. This includes the initial request, so a value of 1 indicates that no requests will be retried. If `total_max_attempts` and `max_attempts` are both provided, `total_max_attempts` takes precedence. `total_max_attempts` is preferred over `max_attempts` because it maps to the `AWS_MAX_ATTEMPTS` environment variable and the `max_attempts` config file value.
- **mode** – A string representing the type of retry mode botocore should use. Valid values are:
  - **legacy** – The default mode. Waits 50ms the first retry, then uses exponential backoff with a base factor of 2. For DynamoDB, it performs up to 10 total max attempts (unless overridden with the above).

    > **Note**
    >
    > With exponential backoff, the last attempt will wait almost 13 seconds.

  - **standard** – Named standard because it's more consistent with other AWS SDKs. Waits a random time from 0ms to 1,000ms for the first retry. If another retry is necessary, it picks another random time from 0ms to 1,000ms and multiplies it by 2. If an additional retry is necessary, it does the same random pick multiplied by 4, and so on. Each wait is capped at 20 seconds. This mode will perform retries on more detected failure conditions than the `legacy` mode. For DynamoDB, it performs up to 3 total max attempts (unless overridden with the above).
  - **adaptive** - An experimental retry mode that includes all the functionality of standard mode but adds automatic client-side throttling. With adaptive rate limiting, SDKs can slow down the rate at which requests are sent to better accommodate the capacity of AWS services. This is a provisional mode whose behavior might change.

An expanded definition of these retry modes can be found in the guide to retries as well as in the Retry behavior topic in the SDK reference.

Here's an example that explicitly uses the `legacy` retry policy with a maximum of 3 total requests (2 retries):

```
import boto3
from botocore.config import Config

my_config = Config(
   connect_timeout = 1.0,
   read_timeout = 1.0,
   retries = {
     'mode': 'legacy',
     'total_max_attempts': 3
   }
)
dynamodb = boto3.resource('dynamodb', config=my_config)
```

Because DynamoDB is a highly-available, low-latency system, you may want to be more aggressive with the speed of retries than the built-in retry policies allow. You can implement your own retry policy by setting the max attempts to 0, catching the exceptions yourself, and retrying as appropriate from your own code instead of relying on boto3 to do implicit retries.

If you manage your own retry policy, you'll want to differentiate between throttles and errors:

- A **throttle** (indicated by a `ProvisionedThroughputExceededException` or `ThrottlingException`) indicates a healthy service that's informing you that you've exceeded your read or write capacity on a DynamoDB table or partition. Every millisecond that passes, a bit more read or write capacity is made available, so you can retry quickly (such as every 50ms) to attempt to access that newly released capacity. With throttles, you don't especially need exponential backoff because throttles are lightweight for DynamoDB to return and incur no per-request charge to you. Exponential backoff assigns longer delays to client threads that have already waited the longest, which statistically extends the p50 and p99 outward.
- An **error** (indicated by an `InternalServerError` or a `ServiceUnavailable`, among others) indicates a transient issue with the service. This can be for the whole table or possibly just the partition you're reading from or writing to. With errors, you can pause longer before retries (such as 250ms or 500ms) and use jitter to stagger the retries.

Config for max pool connections

Lastly, the config lets you control the connection pool size:

- **max_pool_connections (int)** – The maximum number of connections to keep in a connection pool. If this value is not set, the default value of 10 is used.

This option controls the maximum number of HTTP connections to keep pooled for reuse. A different pool is kept per Session. If you anticipate more than 10 threads going against clients or resources built off the same Session, you should consider raising this, so threads don't have to wait on other threads using a pooled connection.

```
import boto3
from botocore.config import Config

my_config = Config(
   max_pool_connections = 20
)

# Setup a single session holding up to 20 pooled connections
session = boto3.Session(my_config)

# Create up to 20 resources against that session for handing to threads
# Notice the single-threaded access to the Session and each Resource
resource1 = session.resource('dynamodb')
resource2 = session.resource('dynamodb')
# etc
```

Error handling

AWS service exceptions aren't all statically defined in Boto3. This is because errors and exceptions from AWS services vary widely and are subject to change. Boto3 wraps all service exceptions as a `ClientError` and exposes the details as structured JSON. For example, an error response might be structured like this:

```
{
    'Error': {
        'Code': 'SomeServiceException',
        'Message': 'Details/context around the exception or error'
    },
    'ResponseMetadata': {
```

```
        'RequestId': '1234567890ABCDEF',
        'HostId': 'host ID data will appear here as a hash',
        'HTTPStatusCode': 400,
        'HTTPHeaders': {'header metadata key/values will appear here'},
        'RetryAttempts': 0
    }
}
```

The following code catches any `ClientError` exceptions and looks at the string value of the `Code` within the `Error` to determine what action to take:

```
import botocore
import boto3

dynamodb = boto3.client('dynamodb')

try:
    response = dynamodb.put_item(...)

except botocore.exceptions.ClientError as err:
    print('Error Code: {}'.format(err.response['Error']['Code']))
    print('Error Message: {}'.format(err.response['Error']['Message']))
    print('Http Code: {}'.format(err.response['ResponseMetadata']['HTTPStatusCode']))
    print('Request ID: {}'.format(err.response['ResponseMetadata']['RequestId']))

    if err.response['Error']['Code'] in ('ProvisionedThroughputExceededException',
 'ThrottlingException'):
        print("Received a throttle")
    elif err.response['Error']['Code'] == 'InternalServerError':
        print("Received a server error")
    else:
        raise err
```

Some (but not all) exception codes have been materialized as top-level classes. You can choose to handle these directly. When using the Client interface, these exceptions are dynamically populated on your client and you catch these exceptions using your client instance, like this:

```
except ddb_client.exceptions.ProvisionedThroughputExceededException:
```

When using the Resource interface, you have to use `.meta.client` to traverse from the resource to the underlying Client to access the exceptions, like this:

```
except ddb_resource.meta.client.exceptions.ProvisionedThroughputExceededException:
```

To review the list of materialized exception types, you can generate the list dynamically:

```
ddb = boto3.client("dynamodb")
print([e for e in dir(ddb.exceptions) if e.endswith('Exception') or
 e.endswith('Error')])
```

When doing a write operation with a condition expression, you can request that if the expression fails the value of the item should be returned in the error response.

```
try:
    response = table.put_item(
        Item=item,
        ConditionExpression='attribute_not_exists(pk)',
        ReturnValuesOnConditionCheckFailure='ALL_OLD'
    )
except table.meta.client.exceptions.ConditionalCheckFailedException as e:
    print('Item already exists:', e.response['Item'])
```

For further reading on error handling and exceptions:

- The boto3 guide on error handling has more information on error handling techniques.
- The DynamoDB developer guide section on programming errors lists what errors you might encounter.
- The Common Errors section in the API reference .
- The documentation on each API operation lists what errors that call might generate (for example BatchWriteItem).

Logging

The boto3 library integrates with Python's built-in logging module for tracking what happens during a session. To control logging levels, you can configure the logging module:

```
import logging

logging.basicConfig(level=logging.INFO)
```

This configures the root logger to log `INFO` and above level messages. Logging messages which are less severe than level will be ignored. Logging levels include `DEBUG`, `INFO`, `WARNING`, `ERROR`, and `CRITICAL`. The default is `WARNING`.

Loggers in boto3 are hierarchical. The library uses a few different loggers, each corresponding to different parts of the library. You can separately control the behavior of each:

- **boto3**: The main logger for the boto3 module.
- **botocore**: The main logger for the botocore package.
- **botocore.auth**: Used for logging AWS signature creation for requests.
- **botocore.credentials**: Used for logging the process of credential fetching and refresh.
- **botocore.endpoint**: Used for logging request creation before it's sent over the network.
- **botocore.hooks**: Used for logging events triggered in the library.
- **botocore.loaders**: Used for logging when parts of AWS service models are loaded.
- **botocore.parsers**: Used for logging AWS service responses before they're parsed.
- **botocore.retryhandler**: Used for logging the processing of AWS service request retries (legacy mode).
- **botocore.retries.standard**: Used for logging the processing of AWS service request retries (standard or adaptive mode).
- **botocore.utils**: Used for logging miscellaneous activities in the library.
- **botocore.waiter**: Used for logging the functionality of waiters, which poll an AWS service until a certain state is reached.

Other libraries log as well. Internally, boto3 uses the third party urllib3 for HTTP connection handling. When latency is important, you can watch its logs to ensure your pool is being well utilized by seeing when urllib3 establishes a new connection or closes an idle one down.

- **urllib3.connectionpool:** Use for logging connection pool handling events.

The following code snippet sets most logging to `INFO` with `DEBUG` logging for endpoint and connection pool activity:

```
import logging

logging.getLogger('boto3').setLevel(logging.INFO)
```

```
logging.getLogger('botocore').setLevel(logging.INFO)
logging.getLogger('botocore.endpoint').setLevel(logging.DEBUG)
logging.getLogger('urllib3.connectionpool').setLevel(logging.DEBUG)
```

Event hooks

Botocore emits events during various parts of its execution. You can register handlers for these events so that whenever an event is emitted, your handler will be called. This lets you extend the behavior of botocore without having to modify the internals.

For instance, let's say you want to keep track of every time a `PutItem` operation is called on any DynamoDB table in your application. You might register on the `'provide-client-params.dynamodb.PutItem'` event to catch and log every time a `PutItem` operation is invoked on the associated Session. Here's an example:

```
import boto3
import botocore
import logging

def log_put_params(params, **kwargs):
    if 'TableName' in params and 'Item' in params:
        logging.info(f"PutItem on table {params['TableName']}: {params['Item']}")

logging.basicConfig(level=logging.INFO)

session = boto3.Session()
event_system = session.events

# Register our interest in hooking in when the parameters are provided to PutItem
event_system.register('provide-client-params.dynamodb.PutItem', log_put_params)

# Now, every time you use this session to put an item in DynamoDB,
# it will log the table name and item data.
dynamodb = session.resource('dynamodb')
table = dynamodb.Table('YourTableName')
table.put_item(
    Item={
        'pk': '123',
        'sk': 'cart#123',
        'item_data': 'YourItemData',
        # ... more attributes ...
    }
```

```
)
```

Within the handler, you can even manipulate the params programmatically to change behavior:

```
params['TableName'] = "NewTableName"
```

For more information on events, see the botocore documentation on events and the boto3 documentation on events.

Pagination and the Paginator

Some requests, such as Query and Scan, limit the size of data returned on a single request and require you to make repeated requests to pull subsequent pages.

You can control the maximum number of items to be read for each page with the `limit` parameter. For example, if you want the last 10 items, you can use `limit` to retrieve only the last 10. Note the limit is how much should be read from the table before any filtering is applied. There's no way to specify you want exactly 10 after filtering; you can only control the pre-filtered count and check client-side when you've actually retrieved 10. Regardless of the limit, every response always has a maximum size of 1 MB.

If the response includes a `LastEvaluatedKey`, it indicates the response ended because it hit a count or size limit. The key is the last key evaluated for the response. You can retrieve this `LastEvaluatedKey` and pass it to a follow-up call as `ExclusiveStartKey` to read the next chunk from that starting point. When there's no `LastEvaluatedKey` returned that, means there are no more items matching the Query or Scan.

Here's a simple example (using the Resource interface, but the Client interface has the same pattern) that reads at most 100 items per page and loops until all items have been read.

```
import boto3

dynamodb = boto3.resource('dynamodb')
table = dynamodb.Table('YourTableName')

query_params = {
    'KeyConditionExpression': Key('pk').eq('123') & Key('sk').gt(1000),
    'Limit': 100
}
```

```
while True:
    response = table.query(**query_params)

    # Process the items however you like
    for item in response['Items']:
        print(item)

    # No LastEvaluatedKey means no more items to retrieve
    if 'LastEvaluatedKey' not in response:
        break

    # If there are possibly more items, update the start key for the next page
    query_params['ExclusiveStartKey'] = response['LastEvaluatedKey']
```

For convenience, boto3 can do this for you with Paginators. However, it only works with the Client interface. Here's the code rewritten to use Paginators:

```
import boto3

dynamodb = boto3.client('dynamodb')

paginator = dynamodb.get_paginator('query')

query_params = {
    'TableName': 'YourTableName',
    'KeyConditionExpression': 'pk = :pk_val AND sk > :sk_val',
    'ExpressionAttributeValues': {
        ':pk_val': {'S': '123'},
        ':sk_val': {'N': '1000'},
    },
    'Limit': 100
}

page_iterator = paginator.paginate(**query_params)

for page in page_iterator:
    # Process the items however you like
    for item in page['Items']:
        print(item)
```

For more information, see the Guide on Paginators and the API reference for DynamoDB.Paginator.Query.

> **Note**
>
> Paginators also have their own configuration settings named `MaxItems`, `StartingToken`, and `PageSize`. For paginating with DynamoDB, you should ignore these settings.

Waiters

Waiters provide the ability to wait for something to complete before proceeding. At present, they only support waiting for a table to be created or deleted. In the background, the waiter operation does a check for you every 20 seconds up to 25 times. You could do this yourself, but using a waiter is elegant when writing automation.

This code shows how to wait for a particular table to have been created:

```
# Create a table, wait until it exists, and print its ARN
response = client.create_table(...)
waiter = client.get_waiter('table_exists')
waiter.wait(TableName='YourTableName')
print('Table created:', response['TableDescription']['TableArn']
```

For more information, see the Guide to Waiters and Reference on Waiters.

Programming Amazon DynamoDB with JavaScript

This guide provides an orientation to programmers wanting to use Amazon DynamoDB with JavaScript. Learn about the AWS SDK for JavaScript, abstraction layers available, configuring connections, handling errors, defining retry policies, managing keep-alive, and more.

Topics

- About AWS SDK for JavaScript
- Using the AWS SDK for JavaScript V3
- Accessing JavaScript documentation
- Abstraction layers
- Using the marshall utility function
- Reading items

- Conditional writes
- Pagination
- Specifying configuration
- Waiters
- Error handling
- Logging
- Considerations

About AWS SDK for JavaScript

The AWS SDK for JavaScript provides access to AWS services using either browser scripts or Node.js. This documentation focuses on the latest version of the SDK (V3). The AWS SDK for JavaScript V3 is maintained by AWS as an open-source project hosted on GitHub. Issues and feature requests are public and you can access them on the issues page for the GitHub repository.

JavaScript V2 is similar to V3, but contains syntax differences. V3 is more modular, making it easier to ship smaller dependencies, and has first-class TypeScript support. We recommend using the latest version of the SDK.

Using the AWS SDK for JavaScript V3

You can add the SDK to your Node.js application using the Node Package Manager. The examples below show how to add the most common SDK packages for working with DynamoDB.

- `npm install @aws-sdk/client-dynamodb`
- `npm install @aws-sdk/lib-dynamodb`
- `npm install @aws-sdk/util-dynamodb`

Installing packages adds references to the dependency section of your package.json project file. You have the option to use the newer ECMAScript module syntax. For further details on these two approaches, see the Considerations section.

Accessing JavaScript documentation

Get started with JavaScript documentation with the following resources:

- Access the Developer guide for core JavaScript documentation. Installation instructions are located in the **Setting up** section.
- Access the API reference documentation to explore all available classes and methods.
- The SDK for JavaScript supports many AWS services other than DynamoDB. Use the following procedure to locate specific API coverage for DynamoDB:

  1. From **Services**, choose **DynamoDB and Libraries**. This documents the low-level client.
  2. Choose **lib-dynamodb**. This documents the high-level client. The two clients represent two different abstraction layers that you have the choice to use. See the section below for more information about abstraction layers.

Abstraction layers

The SDK for JavaScript V3 has a low-level client (`DynamoDBClient`) and a high-level client (`DynamoDBDocumentClient`).

Topics

- Low-level client (DynamoDBClient)
- High-level client (DynamoDBDocumentClient)

Low-level client (`DynamoDBClient`)

The low-level client provides no extra abstractions over the underlying wire protocol. It gives you full control over all aspects of communication, but because there are no abstractions, you must do things like provide item definitions using the DynamoDB JSON format.

As the example below shows, with this format data types must be stated explicitly. An *S* indicates a string value and an *N* indicates a number value. Numbers on the wire are always sent as strings tagged as number types to ensure no loss in precision. The low-level API calls have a naming pattern such as `PutItemCommand` and `GetItemCommand`.

The following example is using low-level client with `Item` defined using DynamoDB JSON:

```
const { DynamoDBClient, PutItemCommand } = require("@aws-sdk/client-dynamodb");

const client = new DynamoDBClient({});

async function addProduct() {
```

```
  const params = {
    TableName: "products",
    Item: {
      "id": { S: "Product01" },
      "description": { S: "Hiking Boots" },
      "category": { S: "footwear" },
      "sku": { S: "hiking-sku-01" },
      "size": { N: "9" }
    }
  };

  try {
    const data = await client.send(new PutItemCommand(params));
    console.log('result : ' + JSON.stringify(data));
  } catch (error) {
    console.error("Error:", error);
  }
}
addProduct();
```

High-level client (`DynamoDBDocumentClient`)

The high-level DynamoDB document client offers built-in convenience features, such as eliminating the need to manually marshal data and allowing for direct reads and writes using standard JavaScript objects. The documentation for `lib-dynamodb` provides the list of advantages.

To instantiate the `DynamoDBDocumentClient`, construct a low-level `DynamoDBClient` and then wrap it with a `DynamoDBDocumentClient`. The function naming convention differs slightly between the two packages. For instance, the low-level uses `PutItemCommand` while the high-level uses `PutCommand`. The distinct names allow both sets of functions to coexist in the same context, allowing you to mix both in the same script.

```
const { DynamoDBClient } = require("@aws-sdk/client-dynamodb");
const { DynamoDBDocumentClient, PutCommand } = require("@aws-sdk/lib-dynamodb");

const client = new DynamoDBClient({});

const docClient = DynamoDBDocumentClient.from(client);

async function addProduct() {
  const params = {
    TableName: "products",
```

```
    Item: {
      id: "Product01",
      description: "Hiking Boots",
      category: "footwear",
      sku: "hiking-sku-01",
      size: 9,
    },
  };

  try {
    const data = await docClient.send(new PutCommand(params));
    console.log('result : ' + JSON.stringify(data));
  } catch (error) {
    console.error("Error:", error);
  }
}

addProduct();
```

The pattern of usage is consistent when you're reading items using API operations such as `GetItem`, `Query`, or `Scan`.

Using the marshall utility function

You can use the low-level client and marshall or unmarshall the data types on your own. The utility package, util-dynamodb, has a `marshall()` utility function that accepts JSON and produces DynamoDB JSON, as well as an `unmarshall()` function, that does the reverse. The following example uses the low-level client with data marshalling handled by the `marshall()` call.

```
const { DynamoDBClient, PutItemCommand } = require("@aws-sdk/client-dynamodb");
const { marshall } = require("@aws-sdk/util-dynamodb");

const client = new DynamoDBClient({});

async function addProduct() {
  const params = {
    TableName: "products",
    Item: marshall({
      id: "Product01",
      description: "Hiking Boots",
      category: "footwear",
      sku: "hiking-sku-01",
```

```
      size: 9,
    }),
  };

  try {
    const data = await client.send(new PutItemCommand(params));
  } catch (error) {
    console.error("Error:", error);
  }
}
addProduct();
```

Reading items

To read a single item from DynamoDB, you use the `GetItem` API operation. Similar to the `PutItem` command, you have the choice to use either the low-level client or the high-level Document client. The example below demonstrates using the high-level Document client to retrieve an item.

```
const { DynamoDBClient } = require("@aws-sdk/client-dynamodb");
const { DynamoDBDocumentClient, GetCommand } = require("@aws-sdk/lib-dynamodb");

const client = new DynamoDBClient({});

const docClient = DynamoDBDocumentClient.from(client);

async function getProduct() {
  const params = {
    TableName: "products",
    Key: {
      id: "Product01",
    },
  };

  try {
    const data = await docClient.send(new GetCommand(params));
    console.log('result : ' + JSON.stringify(data));
  } catch (error) {
    console.error("Error:", error);
  }
}

getProduct();
```

Use the `Query` API operation to read multiple items. You can use the low-level client or the Document client. The example below uses the high-level Document client.

```
const { DynamoDBClient } = require("@aws-sdk/client-dynamodb");
const {
  DynamoDBDocumentClient,
  QueryCommand,
} = require("@aws-sdk/lib-dynamodb");

const client = new DynamoDBClient({});

const docClient = DynamoDBDocumentClient.from(client);

async function productSearch() {
  const params = {
    TableName: "products",
    IndexName: "GSI1",
    KeyConditionExpression: "#category = :category and begins_with(#sku, :sku)",
    ExpressionAttributeNames: {
      "#category": "category",
      "#sku": "sku",
    },
    ExpressionAttributeValues: {
      ":category": "footwear",
      ":sku": "hiking",
    },
  };

  try {
    const data = await docClient.send(new QueryCommand(params));
    console.log('result : ' + JSON.stringify(data));
  } catch (error) {
    console.error("Error:", error);
  }
}

productSearch();
```

Conditional writes

DynamoDB write operations can specify a logical condition expression that must evaluate to true for the write to proceed. If the condition does not evaluate to true, the write operation generates

an exception. The condition expression can check if the item already exists or if its attributes match certain constraints.

```
ConditionExpression = "version = :ver AND size(VideoClip) < :maxsize"
```

When the conditional expression fails, you can use `ReturnValuesOnConditionCheckFailure` to request that the error response include the item that didn't satisfy the conditions to deduce what the problem was. For more details, see Handle conditional write errors in high concurrency scenarios with Amazon DynamoDB.

```
try {
      const response = await client.send(new PutCommand({
          TableName: "YourTableName",
          Item: item,
          ConditionExpression: "attribute_not_exists(pk)",
          ReturnValuesOnConditionCheckFailure: "ALL_OLD"
      }));
  } catch (e) {
      if (e.name === 'ConditionalCheckFailedException') {
          console.log('Item already exists:', e.Item);
      } else {
          throw e;
      }
  }
```

Additional code examples showing other aspects of JavsScript SDK V3 usage are available in the JavaScript SDK V3 Documentation and under the DynamoDB-SDK-Examples GitHub repository.

Pagination

Topics

- Using the paginateScan convenience method

Read requests such as `Scan` or `Query` will likely return multiple items in a dataset. If you perform a `Scan` or `Query` with a `Limit` parameter, then once the system has read that many items, a partial response will be sent, and you'll need to paginate to retrieve additional items.

The system will only read a maximum of 1 megabyte of data per request. If you're including a `Filter` expression, the system will still read a megabyte, at maximum, of data from disk, but will

return the items of that megabyte that match the filter. The filter operation could return 0 items for a page, but still require further pagination before the search is exhausted.

You should look for `LastEvaluatedKey` in the response and using it as the `ExclusiveStartKey` parameter in a subsequent request to continue data retrieval. This serves as a bookmark as noted in the following example.

> **Note**
>
> The sample passes a null `lastEvaluatedKey` as the `ExclusiveStartKey` on the first iteration and this is allowed.

Example using the `LastEvaluatedKey`:

```
const { DynamoDBClient, ScanCommand } = require("@aws-sdk/client-dynamodb");

const client = new DynamoDBClient({});

async function paginatedScan() {
  let lastEvaluatedKey;
  let pageCount = 0;

  do {
    const params = {
      TableName: "products",
      ExclusiveStartKey: lastEvaluatedKey,
    };

    const response = await client.send(new ScanCommand(params));
    pageCount++;
    console.log(`Page ${pageCount}, Items:`, response.Items);
    lastEvaluatedKey = response.LastEvaluatedKey;
  } while (lastEvaluatedKey);
}

paginatedScan().catch((err) => {
  console.error(err);
});
```

Using the `paginateScan` convenience method

The SDK provides convenience methods called `paginateScan` and `paginateQuery` that do this work for you and makes the repeated requests behind the scenes. Specify the max number of items to read per request using the standard `Limit` parameter.

```
const { DynamoDBClient, paginateScan } = require("@aws-sdk/client-dynamodb");

const client = new DynamoDBClient({});

async function paginatedScanUsingPaginator() {
  const params = {
    TableName: "products",
    Limit: 100
  };

  const paginator = paginateScan({client}, params);

  let pageCount = 0;

  for await (const page of paginator) {
    pageCount++;
    console.log(`Page ${pageCount}, Items:`, page.Items);
  }
}

paginatedScanUsingPaginator().catch((err) => {
  console.error(err);
});
```

> **Note**
>
> Performing full table scans regularly is not a recommended access pattern unless the table is small.

Specifying configuration

Topics

- Config for timeouts
- Config for keep-alive
- Config for retries

When setting up the `DynamoDBClient`, you can specify various configuration overrides by passing a configuration object to the constructor. For example, you can specify the Region to connect to if it's not already known to the calling context or the endpoint URL to use. This is useful if you want to target a DynamoDB Local instance for development purposes.

```
const client = new DynamoDBClient({
  region: "eu-west-1",
  endpoint: "http://localhost:8000",
});
```

Config for timeouts

DynamoDB uses HTTPS for client-server communication. You can control some aspects of the HTTP layer by providing a `NodeHttpHandler` object. For example, you can adjust the key timeout values `connectionTimeout` and `requestTimeout`. The `connectionTimeout` is the maximum duration, in milliseconds, that the client will wait while trying to establish a connection before giving up.

The `requestTimeout` defines how long the client will wait for a response after a request has been sent, also in milliseconds. The defaults for both are zero, meaning the timeout is disabled and there's no limit on how long the client will wait if the response does not arrive. You should set the timeouts to something reasonable so in the event of a network issue the request will error out and a new request can be initiated. For example:

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
import { NodeHttpHandler } from "@smithy/node-http-handler";

const requestHandler = new NodeHttpHandler({
  connectionTimeout: 2000,
  requestTimeout: 2000,
});

const client = new DynamoDBClient({
  requestHandler
});
```

> **Note**
>
> The example provided uses the Smithy import. Smithy is a language for defining services and SDKs, open-source and maintained by AWS.

In addition to configuring timeout values, you can set the maximum number of sockets, which allows for an increased number of concurrent connections per origin. The developer guide includes details on configuring the `maxSockets` parameter.

Config for keep-alive

When using HTTPS, the first request always takes some back-and-forth communication to establish a secure connection. HTTP Keep-Alive allows subsequent requests to reuse the already-established connection, making the requests more efficient and lowering latency. HTTP Keep-Alive is enabled by default with JavaScript V3.

There's a limit to how long an idle connection can be kept alive. Consider sending periodic requests, maybe every minute, if you have an idle connection but want the next request to use an already-established connection.

> **Note**
>
> Note that in the older V2 of the SDK, keep-alive was off by default, meaning each connection would get closed immediately after use. If using V2, you can override this setting.

Config for retries

When the SDK receives an error response and the error is resumable as determined by the SDK, such as a throttling exception or a temporary service exception, it will retry again. This happens invisibly to you as the caller, except that you might notice the request took longer to succeed.

The SDK for JavaScript V3 will make 3 total requests, by default, before giving up and passing the error into the calling context. You can adjust the number and frequency of these retries.

The `DynamoDBClient` constructor accepts a `maxAttempts` setting that limits how many attempts will happen. The below example raises the value from the default of 3 to a total of 5. If you set it

to 0 or 1, that indicates you don't want any automatic retries and want to handle any resumable errors yourself within your catch block.

```
const client = new DynamoDBClient({
  maxAttempts: 5,
});
```

You can also control the timing of the retries with a custom retry strategy. To do this, import the `util-retry` utility package and create a custom backoff function that calculates the wait time between retries based on the current retry count.

The example below says to make a maximum of 5 attempts with delays of 15, 30, 90, and 360 milliseconds should the first attempt fail. The custom backoff function, `calculateRetryBackoff`, calculates the delays by accepting the retry attempt number (starts with 1 for the first retry) and returns how many milliseconds to wait for that request.

```
const { ConfiguredRetryStrategy } = require("@aws-sdk/util-retry");

const calculateRetryBackoff = (attempt) => {
  const backoffTimes = [15, 30, 90, 360];
  return backoffTimes[attempt - 1] || 0;
};

const client = new DynamoDBClient({
  retryStrategy: new ConfiguredRetryStrategy(
    5, // max attempts.
    calculateRetryBackoff // backoff function.
  ),
});
```

Waiters

The DynamoDB client includes two useful waiter functions that can be used when creating, modifying, or deleting tables when you want your code to wait to proceed until the table modification has finished. For example, you can deploy a table, call the `waitUntilTableExists` function, and the code will block until the table has been made **ACTIVE**. The waiter internally polls the DynamoDB service with a `describe-table` every 20 seconds.

```
import {waitUntilTableExists, waitUntilTableNotExists} from "@aws-sdk/client-dynamodb";
```

```
… <create table details>

const results = await waitUntilTableExists({client: client, maxWaitTime: 180},
 {TableName: "products"});
if (results.state == 'SUCCESS') {
  return results.reason.Table
}
console.error(`${results.state} ${results.reason}`);
```

The `waitUntilTableExists` feature returns control only when it can perform a `describe-table` command that shows the table status **ACTIVE**. This ensures that you can use `waitUntilTableExists` to wait for the completion of creation, as well as modifications such as adding a GSI index, which may take some time to apply before the table returns to **ACTIVE** status.

Error handling

In the early examples here, we've caught all errors broadly. However, in practical applications, it's important to discern between various error types and implement more precise error handling.

DynamoDB error responses contain metadata, including the name of the error. You can catch errors then match against the possible string names of error conditions to determine how to proceed. For server-side errors, you can leverage the `instanceof` operator with the error types exported by the `@aws-sdk/client-dynamodb` package to manage error handling efficiently.

It's important to note that these errors only manifest after all retries have been exhausted. If an error is retried and is eventually followed by a successful call, from the code's perspective, there's no error just a slightly elevated latency. Retries will show up in Amazon CloudWatch charts as unsuccessful requests, such as throttle or error requests. If the client reaches the maximum retry count, it will give up and generate an exception. This is the client's way of saying it's not going to retry.

Below is a snippet to catch the error and take action based on the type of error that was returned.

```
import {
  ResourceNotFoundException
  ProvisionedThroughputExceededException,
  DynamoDBServiceException,
} from "@aws-sdk/client-dynamodb";

try {
```

```
  await client.send(someCommand);
} catch (e) {
    if (e instanceof ResourceNotFoundException) {
      // Handle ResourceNotFoundException
    } else if (e instanceof ProvisionedThroughputExceededException) {
      // Handle ProvisionedThroughputExceededException
    } else if (e instanceof DynamoDBServiceException) {
      // Handle DynamoDBServiceException
    } else {
      // Other errors such as those from the SDK
      if (e.name === "TimeoutError") {
        // Handle SDK TimeoutError.
      } else {
        // Handle other errors.
      }
    }
}
```

See the section called "Error handling" for common error strings in the *DynamoDB Developer Guide*. The exact errors possible with any particular API call can be found in the documentation for that API call, such as the Query API docs.

The metadata of errors include additional properties, depending on the error. For a `TimeoutError`, the metadata includes the number of attempts that were made and the `totalRetryDelay`, as shown below.

```
{
  "name": "TimeoutError",
  "$metadata": {
    "attempts": 3,
    "totalRetryDelay": 199
  }
}
```

If you manage your own retry policy, you'll want to differentiate between throttles and errors:

- A **throttle** (indicated by a `ProvisionedThroughputExceededException` or `ThrottlingException`) indicates a healthy service that's informing you that you've exceeded your read or write capacity on a DynamoDB table or partition. Every millisecond that passes, a bit more read or write capacity is made available, and so you can retry quickly, such as every 50ms, to attempt to access that newly released capacity.

With throttles you don't especially need exponential backoff because throttles are lightweight for DynamoDB to return and incur no per-request charge to you. Exponential backoff assigns longer delays to client threads that have already waited the longest, which statistically extends the p50 and p99 outward.

- An **error** (indicated by an `InternalServerError` or a `ServiceUnavailable`, among others) indicates a transient issue with the service, possibly the whole table or just the partition you're reading from or writing to. With errors, you can pause longer before retries, such as 250ms or 500ms, and use jitter to stagger the retries.

Logging

Turn on logging to get more details about what the SDK is doing. You can set a parameter on the `DynamoDBClient` as shown in the example below. More log information will appear in the console and includes metadata such as the status code and the consumed capacity. If you run the code locally in a terminal window, the logs appear there. If you run the code in AWS Lambda, and you have Amazon CloudWatch logs set up, then the console output will be written there.

```
const client = new DynamoDBClient({
  logger: console
});
```

You can also hook into the internal SDK activities and perform custom logging as certain events happen. The example below uses the client's `middlewareStack` to intercept each request as it's being sent from the SDK and logs it as it's happening.

```
const client = new DynamoDBClient({});

client.middlewareStack.add(
  (next) => async (args) => {
    console.log("Sending request from AWS SDK", { request: args.request });
    return next(args);
  },
  {
    step: "build",
    name: "log-ddb-calls",
  }
);
```

The `MiddlewareStack` provides a powerful hook for observing and controlling SDK behavior. See the blog Introducing Middleware Stack in Modular AWS SDK for JavaScript, for more information.

Considerations

When implementing the AWS SDK for JavaScript in your project, here are some further factors to consider.

Module systems

The SDK supports two module systems, CommonJS and ES (ECMAScript). CommonJS uses the `require` function, while ES uses the `import` keyword.

1. **Common JS** – `const { DynamoDBClient, PutItemCommand } = require("@aws-sdk/client-dynamodb");`
2. **ES (ECMAScript** – `import { DynamoDBClient, PutItemCommand } from "@aws-sdk/client-dynamodb";`

The project type dictates the module system to be used and is specified in the type section of your package.json file. The default is CommonJS. Use `"type": "module"` to indicate an ES project. If you have an existing Node.JS project that uses the CommonJS package format, you can still add functions with the more modern SDK V3 Import syntax by naming your function files with the .mjs extension. This will allow the code file to be treated as ES (ECMAScript).

Asynchronous operations

You'll see many code samples using callbacks and promises to handle the result of DynamoDB operations. With modern JavaScript this complexity is no longer needed and developers can take advantage of the more succinct and readable async/await syntax for asynchronous operations.

Web browser runtime

Web and mobile developers building with React or React Native can use the SDK for JavaScript in their projects. With the earlier V2 of the SDK, web developers would have to load the full SDK into the browser, referencing an SDK image hosted at https://sdk.amazonaws.com/js/.

With V3, it's possible to bundle just the required V3 client modules and all required JavaScript functions into a single JavaScript file using Webpack, and add it in a script tag in the `<head>` of your HTML pages, as explained in the Getting started in a browser script section of the SDK documentation.

DAX data plane operations

The Amazon DynamoDB Streams Accelerator (DAX) data plane operations are supported by the SDK for JavaScript V3.

Programming DynamoDB with the AWS SDK for Java 2.x

This programming guide provides an orientation for programmers who want to use Amazon DynamoDB with Java. The guide covers different concepts including abstraction layers, configuration management, error handling, controlling retry policies, and managing keep-alive.

Topics

- About the AWS SDK for Java 2.x
- Getting started with the AWS SDK for Java 2.x
- Reviewing the AWS SDK for Java 2.x documentation
- Supported interfaces
- Additional code examples
- Synchronous and asynchronous programming
- HTTP clients
- Configuring an HTTP client
- Error handling
- AWS request ID
- Logging
- Pagination
- Data class annotations

About the AWS SDK for Java 2.x

You can access DynamoDB from Java using the official AWS SDK for Java. The SDK for Java has two versions: 1.x and 2.x. The end-of-support for 1.x was announced on January 12, 2024. It will enter maintenance mode on July 31, 2024 and its end-of-support is due on December 31, 2025. For new development, we highly recommend that you use 2.x, which was first released in 2018. This guide exclusively targets 2.x and focuses only on the parts of the SDK relevant to DynamoDB.

For information about maintenance and support for the AWS SDKs, see AWS SDK and Tools maintenance policy and AWS SDKs and Tools version support matrix in the *AWS SDKs and Tools Reference Guide*.

The AWS SDK for Java 2.x is a major rewrite of the 1.x code base. The SDK for Java 2.x supports modern Java features, such as the non-blocking I/O introduced in Java 8. The SDK for Java 2.x also adds support for pluggable HTTP client implementations to provide more network connection flexibility and configuration options.

A noticeable change between the SDK for Java 1.x and the SDK for Java 2.x is the use of a new package name. The Java 1.x SDK uses the `com.amazonaws` package name, while the Java 2.x SDK uses `software.amazon.awssdk`. Similarly, Maven artifacts for the Java 1.x SDK use the `com.amazonaws` `groupId`, while Java 2.x SDK artifacts use the `software.amazon.awssdk` `groupId`.

> **Important**
>
> The AWS SDK for Java 1.x has a DynamoDB package named `com.amazonaws.dynamodbv2`. The "v2" in the package name doesn't indicate that it's for Java 2 (J2SE). Rather, "v2" indicates that the package supports the second version of the DynamoDB low-level API instead of the original version of the low-level API.

Support for Java versions

The AWS SDK for Java 2.x provides full support for long-term support (LTS) Java releases.

Getting started with the AWS SDK for Java 2.x

The following tutorial shows you how to use Apache Maven for defining dependencies for the SDK for Java 2.x. This tutorial also shows you how to write the code that connects to DynamoDB for listing the available DynamoDB tables. The tutorial in this guide is based on the tutorial Get started with the AWS SDK for Java 2.x in the *AWS SDK for Java 2.x Developer Guide*. We've edited this tutorial to make calls to DynamoDB instead of Amazon S3.

To complete this tutorial, do the following:

- Step 1: Set up for this tutorial
- Step 2: Create the project

- Step 3: Write the code
- Step 4: Build and run the application

Step 1: Set up for this tutorial

Before you begin this tutorial, you need the following:

- Permission to access DynamoDB.
- A Java development environment that's configured with single sign-on access to AWS services using the AWS access portal.

To set up for this tutorial, follow the instructions in Setup overview in the *AWS SDK for Java 2.x Developer Guide*. After you configure your development environment with single sign-on access for the Java SDK and you have an active AWS access portal session, then continue to Step 2 of this tutorial.

Step 2: Create the project

To create the project for this tutorial, you run a Maven command that prompts you for input on how to configure the project. After all input is entered and confirmed, Maven finishes building out the project by creating a `pom.xml` file and creating stub Java files.

1. Open a terminal or command prompt window and navigate to a directory of your choice, for example, your `Desktop` or `Home` folder.
2. Enter the following command at the terminal, and then press **Enter**.

```
mvn archetype:generate \
   -DarchetypeGroupId=software.amazon.awssdk \
   -DarchetypeArtifactId=archetype-app-quickstart \
   -DarchetypeVersion=2.22.0
```

3. For each prompt, enter the value listed in the second column.

Prompt	Value to enter
`Define value for property 'service':`	`dynamodb`

Prompt	Value to enter
`Define value for property 'httpClient' :`	`apache-client`
`Define value for property 'nativeImage' :`	`false`
`Define value for property 'credentialProvider'`	`identity-center`
`Define value for property 'groupId':`	`org.example`
`Define value for property 'artifactId':`	`getstarted`
`Define value for property 'version' 1.0-SNAPSHOT:`	*<Enter>*
`Define value for property 'package' org.example:`	*<Enter>*

4. After you enter the last value, Maven lists the choices that you made. To confirm, enter **Y**. Or, enter **N**, and then re-enter your choices.

Maven creates a project folder named `getstarted` based on the `artifactId` value that you entered. Inside the `getstarted` folder, find a file named `README.md` that you can review, a `pom.xml` file, and a `src` directory.

Maven builds the following directory tree.

```
getstarted
 ### README.md
 ### pom.xml
 ### src
     ### main
     #   ### java
     #   #   ### org
     #   #       ### example
```

```
     #   #           ### App.java
     #   #           ### DependencyFactory.java
     #   #           ### Handler.java
     #   ### resources
     #       ### simplelogger.properties
     ### test
         ### java
             ### org
                 ### example
                     ### HandlerTest.java

 10 directories, 7 files
```

The following shows the contents of the `pom.xml` project file.

`pom.xml`

The `dependencyManagement` section contains a dependency to the AWS SDK for Java 2.x, and the `dependencies` section has a dependency for DynamoDB. Specifying these dependencies forces Maven to include the relevant `.jar` files in your Java class path. By default, the AWS SDK doesn't include all the classes for all AWS services. For DynamoDB, if you use the low-level interface, then you should have a dependency on the `dynamodb` artifact. Or, if you use the high-level interface, on the `dynamodb-enhanced` artifact. If you don't include the relevant dependencies, then your code can't compile. The project uses Java 1.8 because of the `1.8` value in the `maven.compiler.source` and `maven.compiler.target` properties.

```
<?xml version="1.0" encoding="UTF-8"?>
 <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/
XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://
maven.apache.org/xsd/maven-4.0.0.xsd">
     <modelVersion>4.0.0</modelVersion>

     <groupId>org.example</groupId>
     <artifactId>getstarted</artifactId>
     <version>1.0-SNAPSHOT</version>
     <packaging>jar</packaging>
     <properties>
         <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
         <maven.compiler.source>1.8</maven.compiler.source>
         <maven.compiler.target>1.8</maven.compiler.target>
         <maven.shade.plugin.version>3.2.1</maven.shade.plugin.version>
         <maven.compiler.plugin.version>3.6.1</maven.compiler.plugin.version>
```

```
        <exec-maven-plugin.version>1.6.0</exec-maven-plugin.version>
        <aws.java.sdk.version>2.22.0</aws.java.sdk.version> <-------- SDK version
picked up from archetype version.
        <slf4j.version>1.7.28</slf4j.version>
        <junit5.version>5.8.1</junit5.version>
    </properties>

    <dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>software.amazon.awssdk</groupId>
                <artifactId>bom</artifactId>
                <version>${aws.java.sdk.version}</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>
        </dependencies>
    </dependencyManagement>

    <dependencies>
        <dependency>
            <groupId>software.amazon.awssdk</groupId>
            <artifactId>dynamodb</artifactId>  <-------- DynamoDB dependency
            <exclusions>
                <exclusion>
                    <groupId>software.amazon.awssdk</groupId>
                    <artifactId>netty-nio-client</artifactId>
                </exclusion>
                <exclusion>
                    <groupId>software.amazon.awssdk</groupId>
                    <artifactId>apache-client</artifactId>
                </exclusion>
            </exclusions>
        </dependency>

        <dependency>
            <groupId>software.amazon.awssdk</groupId>
            <artifactId>sso</artifactId> <-------- Required for identity center
authentication.
        </dependency>

        <dependency>
            <groupId>software.amazon.awssdk</groupId>
```

```
            <artifactId>ssooidc</artifactId> <-------- Required for identity center
authentication.
        </dependency>

        <dependency>
            <groupId>software.amazon.awssdk</groupId>
            <artifactId>apache-client</artifactId> <-------- HTTP client specified.
            <exclusions>
                <exclusion>
                    <groupId>commons-logging</groupId>
                    <artifactId>commons-logging</artifactId>
                </exclusion>
            </exclusions>
        </dependency>

        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-api</artifactId>
            <version>${slf4j.version}</version>
        </dependency>

        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-simple</artifactId>
            <version>${slf4j.version}</version>
        </dependency>

        <!-- Needed to adapt Apache Commons Logging used by Apache HTTP Client to
Slf4j to avoid
        ClassNotFoundException: org.apache.commons.logging.impl.LogFactoryImpl during
runtime -->
        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>jcl-over-slf4j</artifactId>
            <version>${slf4j.version}</version>
        </dependency>

        <!-- Test Dependencies -->
        <dependency>
            <groupId>org.junit.jupiter</groupId>
            <artifactId>junit-jupiter</artifactId>
            <version>${junit5.version}</version>
            <scope>test</scope>
        </dependency>
```

```
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>${maven.compiler.plugin.version}</version>
            </plugin>
        </plugins>
    </build>

</project>
```

Step 3: Write the code

The following code shows the App class that Maven creates. The `main` method is the entry point into the application, which creates an instance of the `Handler` class and then calls its `sendRequest` method.

App class

```
package org.example;
 import org.slf4j.Logger;
 import org.slf4j.LoggerFactory;

 public class App {
    private static final Logger logger = LoggerFactory.getLogger(App.class);

    public static void main(String... args) {
        logger.info("Application starts");

        Handler handler = new Handler();
        handler.sendRequest();

        logger.info("Application ends");
    }
 }
```

The `DependencyFactory` class that Maven creates contains the `dynamoDbClient` factory method that builds and returns an `DynamoDbClient` instance. The `DynamoDbClient` instance

uses an instance of the Apache-based HTTP client. This is because you specified `apache-client` when Maven prompted you for which HTTP client to use.

The following code shows the `DependencyFactory` class.

DependencyFactory class

```
package org.example;

 import software.amazon.awssdk.http.apache.ApacheHttpClient;
 import software.amazon.awssdk.services.dynamodb.DynamoDbClient;

 /**
  * The module containing all dependencies required by the {@link Handler}.
  */
 public class DependencyFactory {

     private DependencyFactory() {}

     /**
      * @return an instance of DynamoDbClient
      */
     public static DynamoDbClient dynamoDbClient() {
         return DynamoDbClient.builder()
                         .httpClientBuilder(ApacheHttpClient.builder())
                         .build();
     }
 }
```

The `Handler` class contains the main logic of your program. When an instance of `Handler` is created in the App class, the `DependencyFactory` furnishes the `DynamoDbClient` service client. Your code uses the `DynamoDbClient` instance to call DynamoDB.

Maven generates the following `Handler` class with a *`TODO`* comment. The next step in the tutorial replaces the *`TODO`* comment with code.

`Handler` class, Maven-generated

```
package org.example;

 import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
```

```
public class Handler {
    private final DynamoDbClient dynamoDbClient;

    public Handler() {
        dynamoDbClient = DependencyFactory.dynamoDbClient();
    }

    public void sendRequest() {
        // TODO: invoking the API calls using dynamoDbClient.
    }
}
```

To fill in the logic, replace the entire contents of the `Handler` class with the following code. The `sendRequest` method is filled in and the necessary imports are added.

`Handler` class, implemented

The following code uses the `DynamoDbClient` instance to retrieve a list of existing tables. If tables exist for a given account and AWS Region, then the code uses the `Logger` instance to log the names of these tables.

```
package org.example;

 import org.slf4j.Logger;
 import org.slf4j.LoggerFactory;
 import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
 import software.amazon.awssdk.services.dynamodb.model.ListTablesResponse;


 public class Handler {
     private final DynamoDbClient dynamoDbClient;

     public Handler() {
         dynamoDbClient = DependencyFactory.dynamoDbClient();
     }

     public void sendRequest() {
         Logger logger = LoggerFactory.getLogger(Handler.class);

         logger.info("calling the DynamoDB API to get a list of existing tables");
         ListTablesResponse response = dynamoDbClient.listTables();
```

```
        if (!response.hasTableNames()) {
            logger.info("No existing tables found for the configured account &
region");
        } else {
            response.tableNames().forEach(tableName -> logger.info("Table: " +
tableName));
        }
    }
}
```

Step 4: Build and run the application

After you create the project and it contains the complete `Handler` class, build and run the application.

1. Make sure that you have an active AWS IAM Identity Center session. To confirm, run the AWS Command Line Interface (AWS CLI) command `aws sts get-caller-identity` and check the response. If you don't have an active session, then see Sign in using the AWS CLI for instructions.
2. Open a terminal or command prompt window and navigate to your project directory `getstarted`.
3. To build your project, run the following command:

   ```
   mvn clean package
   ```

4. To run the application, run the following command:

   ```
   mvn exec:java -Dexec.mainClass="org.example.App"
   ```

After you view the file, delete the object, and then delete the bucket.

Success

If your Maven project built and ran without error, then congratulations! You've successfully built your first Java application using the SDK for Java 2.x.

Cleanup

To clean up the resources that you created during this tutorial, delete the project folder `getstarted`.

Reviewing the AWS SDK for Java 2.x documentation

The AWS SDK for Java 2.x Developer Guide covers all aspects of the SDK across all AWS services. We recommend that you review the following topics:

- Migrate from version 1.x to 2.x – Includes a detailed explanation of the differences between 1.x and 2.x. This topic also contains instructions about how to use both major versions side-by-side.
- DynamoDB guide for Java 2.x SDK – Shows you how to perform basic DynamoDB operations: creating a table, manipulating items, and retrieving items. These examples use the low-level interface. Java has several interfaces, as explained in the following section: Supported interfaces.

> **Tip**
>
> After you review these topics, bookmark the AWS SDK for Java 2.x API Reference. It covers all AWS services, and we recommend that you use it as your main API reference.

Supported interfaces

The AWS SDK for Java 2.x supports the following interfaces, depending on the level of abstraction that you want.

Topics in this section

- Low-level interface
- High-level interface
- Document interface
- Comparing interfaces with a Query example

Low-level interface

The low-level interface provides a one-to-one mapping to the underlying service API. Every DynamoDB API is available through this interface. This means that the low-level interface can provide complete functionality, but it's often more verbose and complex to use. For example, you have to use the `.s()` functions to hold strings and the `.n()` functions to hold numbers. The following example of PutItem inserts an item using the low-level interface.

```
import org.slf4j.*;
import software.amazon.awssdk.http.crt.AwsCrtHttpClient;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.*;

import java.util.Map;

public class PutItem {

    // Create a DynamoDB client with the default settings connected to the DynamoDB
    // endpoint in the default region based on the default credentials provider chain.
    private static final DynamoDbClient DYNAMODB_CLIENT = DynamoDbClient.create();
    private static final Logger LOGGER = LoggerFactory.getLogger(PutItem.class);

    private void putItem() {
        PutItemResponse response = DYNAMODB_CLIENT.putItem(PutItemRequest.builder()
                .item(Map.of(
                        "pk", AttributeValue.builder().s("123").build(),
                        "sk", AttributeValue.builder().s("cart#123").build(),
                        "item_data",
 AttributeValue.builder().s("YourItemData").build(),
                        "inventory", AttributeValue.builder().n("500").build()
                        // ... more attributes ...
                ))
                .returnConsumedCapacity(ReturnConsumedCapacity.TOTAL)
                .tableName("YourTableName")
                .build());
        LOGGER.info("PutItem call consumed [" +
 response.consumedCapacity().capacityUnits() + "] Write Capacity Unites (WCU)");
    }
}
```

High-level interface

The high-level interface in the AWS SDK for Java 2.x is called the DynamoDB enhanced client. This interface provides a more idiomatic code authoring experience.

The enhanced client offers a way to map between client-side data classes and DynamoDB tables designed to store that data. You define the relationships between tables and their corresponding model classes in your code. Then, you can rely on the SDK to manage the data type manipulation. For more information about the enhanced client, see DynamoDB enhanced client API in the *AWS SDK for Java 2.x Developer Guide*.

The following example of PutItem uses the high-level interface. In this example, the DynamoDbBean named YourItem creates a TableSchema that enables its direct use as input for the putItem( ) call.

```
import org.slf4j.*;
import software.amazon.awssdk.enhanced.dynamodb.*;
import software.amazon.awssdk.enhanced.dynamodb.mapper.annotations.*;
import software.amazon.awssdk.enhanced.dynamodb.model.*;
import software.amazon.awssdk.services.dynamodb.model.ReturnConsumedCapacity;

public class DynamoDbEnhancedClientPutItem {
    private static final DynamoDbEnhancedClient ENHANCED_DYNAMODB_CLIENT =
 DynamoDbEnhancedClient.builder().build();
    private static final DynamoDbTable<YourItem> DYNAMODB_TABLE =
 ENHANCED_DYNAMODB_CLIENT.table("YourTableName", TableSchema.fromBean(YourItem.class));
    private static final Logger LOGGER = LoggerFactory.getLogger(PutItem.class);

    private void putItem() {
        PutItemEnhancedResponse<YourItem> response =
 DYNAMODB_TABLE.putItemWithResponse(PutItemEnhancedRequest.builder(YourItem.class)
                .item(new YourItem("123", "cart#123", "YourItemData", 500))
                .returnConsumedCapacity(ReturnConsumedCapacity.TOTAL)
                .build());
        LOGGER.info("PutItem call consumed [" +
 response.consumedCapacity().capacityUnits() + "] Write Capacity Unites (WCU)");
    }

    @DynamoDbBean
    public static class YourItem {

        public YourItem() {}

        public YourItem(String pk, String sk, String itemData, int inventory) {
            this.pk = pk;
            this.sk = sk;
            this.itemData = itemData;
            this.inventory = inventory;
        }

        private String pk;
        private String sk;
        private String itemData;
```

```
        private int inventory;

        @DynamoDbPartitionKey
        public void setPk(String pk) {
            this.pk = pk;
        }

        public String getPk() {
            return pk;
        }

        @DynamoDbSortKey
        public void setSk(String sk) {
            this.sk = sk;
        }

        public String getSk() {
            return sk;
        }

        public void setItemData(String itemData) {
            this.itemData = itemData;
        }

        public String getItemData() {
            return itemData;
        }

        public void setInventory(int inventory) {
            this.inventory = inventory;
        }

        public int getInventory() {
            return inventory;
        }
    }
}
```

The AWS SDK for Java 1.x has its own high-level interface, which is often referred to by its main class `DynamoDBMapper`. The AWS SDK for Java 2.x is published in a separate package (and Maven artifact) named `software.amazon.awssdk.enhanced.dynamodb`. The Java 2.x SDK is often referred to by its main class `DynamoDbEnhancedClient`.

High-level interface using immutable data classes

The mapping feature of the DynamoDB enhanced client API also works with immutable data classes. An immutable class has only getters and requires a builder class that the SDK uses to create instances of the class. Immutability in Java is a commonly used style that developers can use to create classes that have no side-effects. These classes are more predictable in their behavior in complex multi-threaded applications. Instead of using the `@DynamoDbBean` annotation as shown in the High-level interface example, immutable classes use the `@DynamoDbImmutable` annotation, which takes the builder class as its input.

The following example takes the builder class `DynamoDbEnhancedClientImmutablePutItem` as input to create a table schema. The example then provides the schema as input for the PutItem API call.

```
import org.slf4j.*;
import software.amazon.awssdk.enhanced.dynamodb.*;
import software.amazon.awssdk.enhanced.dynamodb.model.*;
import software.amazon.awssdk.services.dynamodb.model.ReturnConsumedCapacity;

public class DynamoDbEnhancedClientImmutablePutItem {
    private static final DynamoDbEnhancedClient ENHANCED_DYNAMODB_CLIENT =
 DynamoDbEnhancedClient.builder().build();
    private static final DynamoDbTable<YourImmutableItem>
 DYNAMODB_TABLE = ENHANCED_DYNAMODB_CLIENT.table("YourTableName",
 TableSchema.fromImmutableClass(YourImmutableItem.class));
    private static final Logger LOGGER =
 LoggerFactory.getLogger(DynamoDbEnhancedClientImmutablePutItem.class);

    private void putItem() {
        PutItemEnhancedResponse<YourImmutableItem> response =
 DYNAMODB_TABLE.putItemWithResponse(PutItemEnhancedRequest.builder(YourImmutableItem.class)
                .item(YourImmutableItem.builder()
                                    .pk("123")
                                    .sk("cart#123")
                                    .itemData("YourItemData")
                                    .inventory(500)
                                    .build())
                .returnConsumedCapacity(ReturnConsumedCapacity.TOTAL)
                .build());
        LOGGER.info("PutItem call consumed [" +
 response.consumedCapacity().capacityUnits() + "] Write Capacity Unites (WCU)");
    }
```

```
}
```

The following example shows the immutable data class.

```
@DynamoDbImmutable(builder = YourImmutableItem.YourImmutableItemBuilder.class)
class YourImmutableItem {
    private final String pk;
    private final String sk;
    private final String itemData;
    private final int inventory;
    public YourImmutableItem(YourImmutableItemBuilder builder) {
        this.pk = builder.pk;
        this.sk = builder.sk;
        this.itemData = builder.itemData;
        this.inventory = builder.inventory;
    }

    public static YourImmutableItemBuilder builder() { return new
 YourImmutableItemBuilder(); }

    @DynamoDbPartitionKey
    public String getPk() {
        return pk;
    }

    @DynamoDbSortKey
    public String getSk() {
        return sk;
    }

    public String getItemData() {
        return itemData;
    }

    public int getInventory() {
        return inventory;
    }

    static final class YourImmutableItemBuilder {
        private String pk;
        private String sk;
        private String itemData;
        private int inventory;
```

```
        private YourImmutableItemBuilder() {}

        public YourImmutableItemBuilder pk(String pk) { this.pk = pk; return this; }
        public YourImmutableItemBuilder sk(String sk) { this.sk = sk; return this; }
        public YourImmutableItemBuilder itemData(String itemData) { this.itemData =
 itemData; return this; }
        public YourImmutableItemBuilder inventory(int inventory) { this.inventory =
 inventory; return this; }

        public YourImmutableItem build() { return new YourImmutableItem(this); }
    }
}
```

High-level interface using immutable data classes and third-party boilerplate generation libraries

Immutable data classes (shown in the previous example) require some boilerplate code. For example, the getter and setter logic on the data classes, in addition to the `Builder` classes. Third-party libraries, such as Project Lombok, can help you generate that type of boilerplate code. Reducing most of the boilerplate code helps you limit the amount of code needed for working with immutable data classes and the AWS SDK. This further results in improved productivity and readability of your code. For more information, see Use third-party libraries, such as Lombok in the *AWS SDK for Java 2.x Developer Guide*.

The following example demonstrates how Project Lombok simplifies the code needed to use the DynamoDB enhanced client API.

```
import org.slf4j.*;
import software.amazon.awssdk.enhanced.dynamodb.*;
import software.amazon.awssdk.enhanced.dynamodb.model.*;
import software.amazon.awssdk.services.dynamodb.model.ReturnConsumedCapacity;

public class DynamoDbEnhancedClientImmutableLombokPutItem {

    private static final DynamoDbEnhancedClient ENHANCED_DYNAMODB_CLIENT =
 DynamoDbEnhancedClient.builder().build();
    private static final DynamoDbTable<YourImmutableLombokItem>
 DYNAMODB_TABLE = ENHANCED_DYNAMODB_CLIENT.table("YourTableName",
 TableSchema.fromImmutableClass(YourImmutableLombokItem.class));
    private static final Logger LOGGER =
 LoggerFactory.getLogger(DynamoDbEnhancedClientImmutableLombokPutItem.class);
```

```
    private void putItem() {
        PutItemEnhancedResponse<YourImmutableLombokItem> response =
 DYNAMODB_TABLE.putItemWithResponse(PutItemEnhancedRequest.builder(YourImmutableLombokItem.clas
                .item(YourImmutableLombokItem.builder()
                        .pk("123")
                        .sk("cart#123")
                        .itemData("YourItemData")
                        .inventory(500)
                        .build())
                .returnConsumedCapacity(ReturnConsumedCapacity.TOTAL)
                .build());
        LOGGER.info("PutItem call consumed [" +
 response.consumedCapacity().capacityUnits() + "] Write Capacity Unites (WCU)");
    }
}
```

The following example shows the immutable data object of the immutable data class.

```
import lombok.*;
import software.amazon.awssdk.enhanced.dynamodb.mapper.annotations.*;

@Builder
@DynamoDbImmutable(builder =
 YourImmutableLombokItem.YourImmutableLombokItemBuilder.class)
@Value
public class YourImmutableLombokItem {

    @Getter(onMethod_=@DynamoDbPartitionKey)
    String pk;
    @Getter(onMethod_=@DynamoDbSortKey)
    String sk;
    String itemData;
    int inventory;
}
```

The `YourImmutableLombokItem` class uses the following annotations that Project Lombok and the AWS SDK provide:

- `@Builder` – Produces complex builder APIs for data classes that Project Lombok provides.
- `@DynamoDbImmutable` – Identifies the `DynamoDbImmutable` class as a DynamoDB mappable entity annotation that the AWS SDK provides.

- @Value – The immutable variant of `@Data`. By default, all fields are made private and final, and setters are not generated. Project Lombok provides this annotation.

Document interface

The AWS SDK for Java 2.x Document interface avoids the need to specify data type descriptors. The data types are implied by the semantics of the data itself. This Document interface is similar to the AWS SDK for Java 1.x, Document interface, but with a redesigned interface.

The following Document interface example shows the `PutItem` call expressed using the Document interface. The example also uses EnhancedDocument. To run commands against a DynamoDB table using the enhanced document API, you must first associate the table with your document table schema to create a `DynamoDBTable` resource object. The Document table schema builder requires the primary index key and attribute converter providers.

You can use `AttributeConverterProvider.defaultProvider()` to convert document attributes of default types. You can change the overall default behavior with a custom `AttributeConverterProvider` implementation. You can also change the converter for a single attribute. The AWS SDKs and Tools Reference Guide provides more details and examples about how to use custom converters. Their primary use is for attributes of your domain classes that don't have a default converter available. Using a custom converter, you can provide the SDK with the needed information to write or read to DynamoDB.

```
import org.slf4j.*;
import software.amazon.awssdk.enhanced.dynamodb.*;
import software.amazon.awssdk.enhanced.dynamodb.document.EnhancedDocument;
import software.amazon.awssdk.enhanced.dynamodb.model.*;
import software.amazon.awssdk.services.dynamodb.model.ReturnConsumedCapacity;

public class DynamoDbEnhancedDocumentClientPutItem {
    private static final DynamoDbEnhancedClient ENHANCED_DYNAMODB_CLIENT =
 DynamoDbEnhancedClient.builder().build();
    private static final DynamoDbTable<EnhancedDocument> DYNAMODB_TABLE =
            ENHANCED_DYNAMODB_CLIENT.table("YourTableName",
 TableSchema.documentSchemaBuilder()

 .addIndexPartitionKey(TableMetadata.primaryIndexName(),"pk", AttributeValueType.S)
                            .addIndexSortKey(TableMetadata.primaryIndexName(), "sk",
 AttributeValueType.S)
```

```
 .attributeConverterProviders(AttributeConverterProvider.defaultProvider())
                             .build());

    private static final Logger LOGGER =
 LoggerFactory.getLogger(DynamoDbEnhancedDocumentClientPutItem.class);

    private void putItem() {
        PutItemEnhancedResponse<EnhancedDocument> response =
 DYNAMODB_TABLE.putItemWithResponse(
                         PutItemEnhancedRequest.builder(EnhancedDocument.class)
                                 .item(
                                     EnhancedDocument.builder()

 .attributeConverterProviders(AttributeConverterProvider.defaultProvider())
                                             .putString("pk", "123")
                                             .putString("sk", "cart#123")
                                             .putString("item_data", "YourItemData")
                                             .putNumber("inventory", 500)
                                             .build())
                                 .returnConsumedCapacity(ReturnConsumedCapacity.TOTAL)
                                 .build());
        LOGGER.info("PutItem call consumed [" +
 response.consumedCapacity().capacityUnits() + "] Write Capacity Unites (WCU)");
    }

}
```

To convert JSON documents to and from the native Amazon DynamoDB data types, you can use the following utility methods:

- `EnhancedDocument.fromJson(String json)` – Creates a new EnhancedDocument instance from a JSON string.
- `EnhancedDocument.toJson()` – Creates a JSON string representation of the document that you can use in your application like any other JSON object.

Comparing interfaces with a `Query` example

This section shows the same `Query` call expressed using the various interfaces. To fine tune the results of these queries, note the following:

- DynamoDB targets one specific partition key value, so you must specify the partition key completely.
- To have the query target only cart items, the sort key has a key condition expression that uses `begins_with`.
- We use `limit()` to limit the query to a maximum of 100 returned items.
- We set the `scanIndexForward` to false. The results are returned in order of UTF-8 bytes, which usually means the cart item with the lowest number is returned first. By setting the `scanIndexForward` to false, we reverse the order and the cart item with the highest number is returned first.
- We apply a filter to remove any result that does not match the criteria. The data being filtered consumes read capacity whether the item matches the filter.

Example `Query` using the low-level interface

The following example queries a table named `YourTableName` using a `keyConditionExpression`. This limits the query to a specific partition key value and sort key value that begin with a specific prefix value. These key conditions limit the amount of data read from DynamoDB. Finally, the query applies a filter on the data retrieved from DynamoDB using a `filterExpression`.

```
import org.slf4j.*;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.*;

import java.util.Map;

public class Query {

    // Create a DynamoDB client with the default settings connected to the DynamoDB
    // endpoint in the default region based on the default credentials provider chain.
    private static final DynamoDbClient DYNAMODB_CLIENT =
 DynamoDbClient.builder().build();
    private static final Logger LOGGER = LoggerFactory.getLogger(Query.class);

    private static void query() {
        QueryResponse response = DYNAMODB_CLIENT.query(QueryRequest.builder()
                .expressionAttributeNames(Map.of("#name", "name"))
                .expressionAttributeValues(Map.of(
                    ":pk_val", AttributeValue.fromS("id#1"),
```

```
                    ":sk_val", AttributeValue.fromS("cart#"),
                    ":name_val", AttributeValue.fromS("SomeName")))
                .filterExpression("#name = :name_val")
                .keyConditionExpression("pk = :pk_val AND begins_with(sk, :sk_val)")
                .limit(100)
                .scanIndexForward(false)
                .tableName("YourTableName")
                .build());

        LOGGER.info("nr of items: " + response.count());
        LOGGER.info("First item pk: " + response.items().get(0).get("pk"));
        LOGGER.info("First item sk: " + response.items().get(0).get("sk"));
    }
}
```

Example `Query` using the Document interface

The following example queries a table named `YourTableName` using the Document interface.

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import software.amazon.awssdk.enhanced.dynamodb.*;
import software.amazon.awssdk.enhanced.dynamodb.document.EnhancedDocument;
import software.amazon.awssdk.enhanced.dynamodb.model.*;

import java.util.Map;

public class DynamoDbEnhancedDocumentClientQuery {

    // Create a DynamoDB client with the default settings connected to the DynamoDB
    // endpoint in the default region based on the default credentials provider chain.
    private static final DynamoDbEnhancedClient ENHANCED_DYNAMODB_CLIENT =
 DynamoDbEnhancedClient.builder().build();
    private static final DynamoDbTable<EnhancedDocument> DYNAMODB_TABLE =
            ENHANCED_DYNAMODB_CLIENT.table("YourTableName",
 TableSchema.documentSchemaBuilder()
                    .addIndexPartitionKey(TableMetadata.primaryIndexName(),"pk",
 AttributeValueType.S)
                    .addIndexSortKey(TableMetadata.primaryIndexName(), "sk",
 AttributeValueType.S)

 .attributeConverterProviders(AttributeConverterProvider.defaultProvider())
                    .build());
```

```
    private static final Logger LOGGER =
 LoggerFactory.getLogger(DynamoDbEnhancedDocumentClientQuery.class);

    private void query() {
        PageIterable<EnhancedDocument> response =
 DYNAMODB_TABLE.query(QueryEnhancedRequest.builder()
                .filterExpression(Expression.builder()
                        .expression("#name = :name_val")
                        .expressionNames(Map.of("#name", "name"))
                        .expressionValues(Map.of(":name_val",
 AttributeValue.fromS("SomeName")))
                        .build())
                .limit(100)
                .queryConditional(QueryConditional.sortBeginsWith(Key.builder()
                        .partitionValue("id#1")
                        .sortValue("cart#")
                        .build()))
                .scanIndexForward(false)
                .build());

        LOGGER.info("nr of items: " + response.items().stream().count());
        LOGGER.info("First item pk: " +
 response.items().iterator().next().getString("pk"));
        LOGGER.info("First item sk: " +
 response.items().iterator().next().getString("sk"));

    }
}
```

Example `Query` using the high-level interface

The following example queries a table named `YourTableName` using the DynamoDB enhanced client API.

```
import org.slf4j.*;
import software.amazon.awssdk.enhanced.dynamodb.*;
import software.amazon.awssdk.enhanced.dynamodb.mapper.annotations.*;
import software.amazon.awssdk.enhanced.dynamodb.model.*;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;

import java.util.Map;

public class DynamoDbEnhancedClientQuery {
```

```
   private static final DynamoDbEnhancedClient ENHANCED_DYNAMODB_CLIENT =
DynamoDbEnhancedClient.builder().build();
   private static final DynamoDbTable<YourItem> DYNAMODB_TABLE =
ENHANCED_DYNAMODB_CLIENT.table("YourTableName",
TableSchema.fromBean(DynamoDbEnhancedClientQuery.YourItem.class));
   private static final Logger LOGGER =
LoggerFactory.getLogger(DynamoDbEnhancedClientQuery.class);

   private void query() {
       PageIterable<YourItem> response =
DYNAMODB_TABLE.query(QueryEnhancedRequest.builder()
               .filterExpression(Expression.builder()
                       .expression("#name = :name_val")
                       .expressionNames(Map.of("#name", "name"))
                       .expressionValues(Map.of(":name_val",
AttributeValue.fromS("SomeName")))
                       .build())
               .limit(100)
               .queryConditional(QueryConditional.sortBeginsWith(Key.builder()
                       .partitionValue("id#1")
                       .sortValue("cart#")
                       .build()))
               .scanIndexForward(false)
               .build());

       LOGGER.info("nr of items: " + response.items().stream().count());
       LOGGER.info("First item pk: " + response.items().iterator().next().getPk());
       LOGGER.info("First item sk: " + response.items().iterator().next().getSk());
   }

   @DynamoDbBean
   public static class YourItem {

       public YourItem() {}

       public YourItem(String pk, String sk, String name) {
           this.pk = pk;
           this.sk = sk;
           this.name = name;
       }

       private String pk;
       private String sk;
```

```
        private String name;

        @DynamoDbPartitionKey
        public void setPk(String pk) {
            this.pk = pk;
        }

        public String getPk() {
            return pk;
        }

        @DynamoDbSortKey
        public void setSk(String sk) {
            this.sk = sk;
        }

        public String getSk() {
            return sk;
        }

        public void setName(String name) {
            this.name = name;
        }

        public String getName() {
            return name;
        }
    }
}
```

High-level interface using immutable data classes

When you perform a `Query` with the high-level immutable data classes, the code is the same as the high-level interface example except for the construction of the entity class `YourItem` or `YourImmutableItem`. For more information, see the PutItem example.

High-level interface using immutable data classes and third-party boilerplate generation libraries

When you perform a `Query` with the high-level immutable data classes, the code is the same as the high-level interface example except for the construction of the entity class `YourItem` or `YourImmutableLombokItem`. For more information, see the PutItem example.

Additional code examples

For additional examples of how to use DynamoDB with the SDK for Java 2.x, refer to the following code example repositories:

- Official AWS single-action code examples
- Community-maintained single-action code examples
- Official AWS scenario-oriented code examples

Synchronous and asynchronous programming

The AWS SDK for Java 2.x provides both *synchronous* and *asynchronous* clients for AWS services, such as DynamoDB.

The `DynamoDbClient` and `DynamoDbEnhancedClient` classes provide synchronous methods that block your thread's execution until the client receives a response from the service. This client is the most straightforward way of interacting with DynamoDB if you have no need for asynchronous operations.

The `DynamoDbAsyncClient` and `DynamoDbEnhancedAsyncClient` classes provide asynchronous methods that return immediately, and give control back to the calling thread without waiting for a response. The non-blocking client has an advantage that it uses for high concurrency across a few threads, which provides efficient handling of I/O requests with minimal compute resources. This improves throughput and responsiveness.

The AWS SDK for Java 2.x uses the native support for non-blocking I/O. The AWS SDK for Java 1.x had to simulate non-blocking I/O.

The synchronous methods return before a response is available, so you need a way to get the response when it's ready. The asynchronous methods in the AWS SDK for Java return a `CompletableFuture` object that contains the results of the asynchronous operation in the future. When you call `get()` or `join()` on these `CompletableFuture` objects, your code blocks until the result is available. If you call these at the same time that you make the request, then the behavior is similar to a plain synchronous call.

For more information about asynchronous programming, see Use asynchronous programming in the *AWS SDK for Java 2.x Developer Guide*.

HTTP clients

For supporting every client, there exists an HTTP client that handles communication with the AWS services. You can plug in alternative HTTP clients, choosing one that has the characteristics that best fit your application. Some are more lightweight; some have more configuration options.

Some HTTP clients support only synchronous use, while others support only asynchronous use. For a flowchart that can help you select the optimal HTTP client for your workload, see HTTP client recommendations in the *AWS SDK for Java 2.x Developer Guide*.

The following list presents some of the possible HTTP clients:

Topics

- Apache-based HTTP client
- URLConnection-based HTTP client
- Netty-based HTTP client
- AWS CRT-based HTTP client

Apache-based HTTP client

The `ApacheHttpClient` class supports synchronous service clients. It's the default HTTP client for synchronous use. For information about configuring the `ApacheHttpClient` class, see Configure the Apache-based HTTP client in the *AWS SDK for Java 2.x Developer Guide*.

URLConnection-based HTTP client

The `UrlConnectionHttpClient` class is another option for synchronous clients. It loads more quickly than the Apache-based HTTP client, but has fewer features. For information about configuring the `UrlConnectionHttpClient` class, see Configure the URLConnection-based HTTP client in the *AWS SDK for Java 2.x Developer Guide*.

Netty-based HTTP client

The `NettyNioAsyncHttpClient` class supports async clients. It's the default choice for async use. For information about configuring the `NettyNioAsyncHttpClient` class, see Configure the Netty-based HTTP client in the *AWS SDK for Java 2.x Developer Guide*.

AWS CRT-based HTTP client

The newer `AwsCrtHttpClient` and `AwsCrtAsyncHttpClient` classes from the AWS Common Runtime (CRT) libraries are more options that support synchronous and asynchronous clients. Compared to other HTTP clients, AWS CRT offers:

- Faster SDK startup time
- Smaller memory footprint
- Reduced latency time
- Connection health management
- DNS load balancing

For information about configuring the `AwsCrtHttpClient` and `AwsCrtAsyncHttpClient` classes, see Configure the AWS CRT-based HTTP clients in the *AWS SDK for Java 2.x Developer Guide*.

The AWS CRT-based HTTP client isn't the default because that would break backward compatibility for existing applications. However, for DynamoDB we recommend that you use the AWS CRT-based HTTP client for both sync and async uses.

For an introduction to the AWS CRT-based HTTP client, see Announcing availability of the AWS CRT HTTP Client in the AWS SDK for Java 2.x on the *AWS Developer Tools Blog*.

Configuring an HTTP client

When configuring a client, you can provide various configuration options, including:

- Setting timeouts for different aspects of API calls.
- Enabling TCP Keep-Alive.
- Controlling the retry policy when encountering errors.
- Specifying execution attributes that Execution interceptor instances can modify. Execution interceptors can write code that intercept the execution of your API requests and responses. This enables you to perform tasks such as publishing metrics and modifying requests in-flight.
- Adding or manipulating HTTP headers.
- Enabling the tracking of client-side performance metrics. Using this feature helps you to collect metrics about the service clients in your application and analyze the output in Amazon CloudWatch.

- Specifying an alternate executor service to be used for scheduling tasks, such as async retry attempts and timeout tasks.

You control the configuration by providing a `ClientOverrideConfiguration` object to the service client `Builder` class. You'll see this in some code examples in the following sections.

The `ClientOverrideConfiguration` provides standard configuration choices. The different pluggable HTTP clients have implementation-specific configuration possibilities as well.

Topics in this section

- Timeout configuration
- RetryMode
- DefaultsMode
- Keep-Alive configuration

Timeout configuration

You can adjust the client configuration to control various timeouts related to the service calls. DynamoDB provides lower latencies compared to other AWS services. Therefore, you might want to adjust these properties to lower timeout values so that you can fail fast if there's a networking issue.

You can customize the latency related behavior using `ClientOverrideConfiguration` on the DynamoDB client or by changing detailed configuration options on the underlying HTTP client implementation.

You can configure the following impactful properties using `ClientOverrideConfiguration`:

- `apiCallAttemptTimeout` – The amount of time to wait for a single attempt for an HTTP request to complete before giving up and timing out.
- `apiCallTimeout` – The amount of time that the client has to completely execute an API call. This includes the request handler execution that consists of all the HTTP requests, including retries.

The AWS SDK for Java 2.x provides default values for some timeout options, such as connection timeout and socket timeouts. The SDK doesn't provide default values for API call timeouts or individual API call attempt timeouts. If these timeouts aren't set in the

`ClientOverrideConfiguration`, then the SDK uses the socket timeout value instead for the overall API call timeout. The socket timeout has a default value of 30 seconds.

RetryMode

Another configuration related to the timeout configuration that you should consider is the `RetryMode` configuration object. This configuration object contains a collection of retry behaviors.

The SDK for Java 2.x supports the following retry modes:

- `legacy` – The default retry mode if you don't explicitly change it. This retry mode is specific to the Java SDK. It's characterized by up to three retries, or more for services such as DynamoDB, which has up to eight retries.
- `standard` – Named "standard" because it's more consistent with other AWS SDKs. This mode waits for a random amount of time ranging from 0ms to 1,000ms for the first retry. If another retry is necessary, then this mode picks another random time from 0ms to 1,000ms and multiplies it by two. If an additional retry is necessary, then it does the same random pick multiplied by four, and so on. Each wait is capped at 20 seconds. This mode performs retries on more detected failure conditions than the `legacy` mode. For DynamoDB, it performs up to three total max attempts unless you override with numRetries.
- `adaptive` – Builds on `standard` mode and dynamically limits the rate of AWS requests to maximize success rate. This can occur at the expense of request latency. We don't recommend adaptive retry mode when predictable latency is important.

You can find an expanded definition of these retry modes in the Retry behavior topic in the *AWS SDKs and Tools Reference Guide*.

Retry policies

All `RetryMode` configurations have a `RetryPolicy`, which is built based on one or more `RetryCondition` configurations. The `TokenBucketRetryCondition` is especially important to the retry behavior of the DynamoDB SDK client implementation. This condition limits the number of retries that the SDK makes using a token bucket algorithm. Depending on the selected retry mode, throttling exceptions may or may not subtract tokens from the `TokenBucket`.

When a client encounters a retryable error, such as a throttling exception or a temporary server error, then the SDK automatically retries the request. You can control how many times and how quickly these retries happen.

When configuring a client, you can provide a `RetryPolicy` that supports the following parameters:

- `numRetries` – The maximum number of retries that should be applied before a request is considered to be failed. The default value is 8 regardless of the retry mode that you use.

  > **Warning**
  >
  > Make sure that you change this default value after due consideration.

- `backoffStrategy` – The `BackoffStrategy` to apply to the retries, with `FullJitterBackoffStrategy` being the default strategy. This strategy performs an exponential delay between additional retries based on the current number or retries, a base delay, and a maximum backoff time. It then adds jitter to provide a bit of randomness. The base delay used in the exponential delay is 25 ms regardless of the retry mode.
- `retryCondition` – The `RetryCondition` determines whether to retry a request at all. By default, it retries a specific set of HTTP status codes and exceptions that it believes are retryable. For most situations, the default configuration should be sufficient.

The following code provides an alternative retry policy. It specifies a total of five retries (six total requests). The first retry should occur after a delay of approximately 100ms, with each additional retry doubling that time exponentially, up to a maximum delay of one second.

```
DynamoDbClient client = DynamoDbClient.builder()
    .overrideConfiguration(ClientOverrideConfiguration.builder()
        .retryPolicy(RetryPolicy.builder()
            .backoffStrategy(FullJitterBackoffStrategy.builder()
                .baseDelay(Duration.ofMillis(100))
                .maxBackoffTime(Duration.ofSeconds(1))
                .build())
            .numRetries(5)
            .build())
        .build())
    .build();
```

DefaultsMode

The timeout properties that `ClientOverrideConfiguration` and the `RetryMode` don't manage are typically configured implicitly by specifying a `DefaultsMode`.

The AWS SDK for Java 2.x (version 2.17.102 or later) introduced support for `DefaultsMode`. This feature provides a set of default values for common configurable settings, such as HTTP communication settings, retry behavior, service Regional endpoint settings, and potentially any SDK-related configuration. When you use this feature, you can get new configuration defaults tailored to common usage scenarios.

The default modes are standardized across all of the AWS SDKs. The SDK for Java 2.x supports the following default modes:

- `legacy` – Provides default settings that vary by AWS SDK and that existed before `DefaultsMode` was established.
- `standard` – Provides default non-optimized settings for most scenarios.
- `in-region` – Builds on the standard mode and includes settings tailored for applications that call AWS services from within the same AWS Region.
- `cross-region` – Builds on the standard mode and includes settings with high timeouts for applications that call AWS services in a different Region.
- `mobile` – Builds on the standard mode and includes settings with high timeouts tailored for mobile applications with higher latencies.
- `auto` – Builds on the standard mode and includes experimental features. The SDK attempts to discover the runtime environment to determine the appropriate settings automatically. The auto-detection is heuristics-based and does not provide 100% accuracy. If the runtime environment can't be determined, then standard mode is used. The auto-detection might query Instance metadata and user data, which might introduce latency. If startup latency is critical to your application, we recommend choosing an explicit `DefaultsMode` instead.

You can configure the defaults mode in the following ways:

- Directly on a client, through `AwsClientBuilder.Builder#defaultsMode(DefaultsMode)`.
- On a configuration profile, through the `defaults_mode` profile file property.
- Globally, through the `aws.defaultsMode` system property.
- Globally, through the `AWS_DEFAULTS_MODE` environment variable.

> **Note**
>
> For any mode other than `legacy`, the vended default values might change as best practices evolve. Therefore, if you're using a mode other than `legacy`, then we encourage you to perform testing when upgrading the SDK.

The Smart configuration defaults in the *AWS SDKs and Tools Reference Guide* provides a list of configuration properties and their default values in the different default modes.

You choose the defaults mode value based on your application's characteristics and the AWS service that the application interacts with.

These values are configured with a broad selection of AWS services in mind. For a typical DynamoDB deployment in which both your DynamoDB tables and application are deployed in one Region, the `in-region` defaults mode is most relevant among the `standard` default modes.

Example DynamoDB SDK client configuration tuned for low-latency calls

The following example adjusts the timeouts to lower values for an expected low-latency DynamoDB call.

```
DynamoDbAsyncClient asyncClient = DynamoDbAsyncClient.builder()
    .defaultsMode(DefaultsMode.IN_REGION)
    .httpClientBuilder(AwsCrtAsyncHttpClient.builder())
    .overrideConfiguration(ClientOverrideConfiguration.builder()
        .apiCallTimeout(Duration.ofSeconds(3))
        .apiCallAttemptTimeout(Duration.ofMillis(500))
        .build())
    .build();
```

The individual HTTP client implementation may provide you with even more granular control over the timeout and connection usage behavior. For example, for the AWS CRT-based client, you can enable `ConnectionHealthConfiguration`, which enables the client to actively monitor the health of the used connections. For more information, see Advanced configuration of AWS CRT-based HTTP clients in the *AWS SDK for Java 2.x Developer Guide*.

Keep-Alive configuration

Enabling keep-alive can reduce latencies by reusing connections. There are two different kinds of keep-alive: HTTP Keep-Alive and TCP Keep-Alive.

- HTTP Keep-Alive attempts to maintain the HTTPS connection between the client and server so later requests can reuse that connection. This skips the heavyweight HTTPS authentication on later requests. HTTP Keep-Alive is enabled by default on all clients.
- TCP Keep-Alive requests that the underlying operating system sends small packets over the socket connection to provide extra assurance that the socket is kept alive and to immediately detect any drops. This ensures that a later request won't spend time trying to use a dropped socket. By default, TCP Keep-Alive is disabled on all clients. The following code examples show how to enable it on each HTTP client. When enabled for all non-CRT based HTTP clients, the actual Keep-Alive mechanism is dependent on the operating system. Therefore, you must configure additional TCP Keep-Alive values, such as timeout and number of packets, through the operating system. You can do this using `sysctl` on Linux or macOS, or using registry values on Windows.

Example to enable TCP Keep-Alive on an Apache-based HTTP client

```
DynamoDbClient client = DynamoDbClient.builder()
    .httpClientBuilder(ApacheHttpClient.builder().tcpKeepAlive(true))
    .build();
```

`URLConnection`-based HTTP client

Any synchronous client that uses the `URLConnection`-based HTTP client `HttpURLConnection` doesn't have a mechanism to enable keep-alive.

Example to enable TCP Keep-Alive on a Netty-based HTTP client

```
DynamoDbAsyncClient client = DynamoDbAsyncClient.builder()
    .httpClientBuilder(NettyNioAsyncHttpClient.builder().tcpKeepAlive(true))
    .build();
```

Example to enable TCP Keep-Alive on an AWS CRT-based HTTP client

With the AWS CRT-based HTTP client, you can enable TCP keep-alive and control the duration.

```
DynamoDbClient client = DynamoDbClient.builder()
    .httpClientBuilder(AwsCrtHttpClient.builder()
    .tcpKeepAliveConfiguration(TcpKeepAliveConfiguration.builder()
        .keepAliveInterval(Duration.ofSeconds(50))
        .keepAliveTimeout(Duration.ofSeconds(5))
```

```
        .build()))
    .build();
```

When using the asynchronous DynamoDB client, you can enable TCP Keep-Alive as shown in the following code.

```
DynamoDbAsyncClient client = DynamoDbAsyncClient.builder()
    .httpClientBuilder(AwsCrtAsyncHttpClient.builder()
    .tcpKeepAliveConfiguration(TcpKeepAliveConfiguration.builder()
        .keepAliveInterval(Duration.ofSeconds(50))
        .keepAliveTimeout(Duration.ofSeconds(5))
        .build()))
    .build();
```

Error handling

When it comes to exception handling, the AWS SDK for Java 2.x uses runtime (unchecked) exceptions.

The base exception, covering all SDK exceptions, is `SdkServiceException`, which extends from the Java unchecked `RuntimeException`. If you catch this, you'll catch all exceptions that the SDK throws.

`SdkServiceException` has a subclass called `AwsServiceException`. This subclass indicates any issue in communication with the AWS service. It has a subclass called `DynamoDbException`, which indicates an issue in communication with DynamoDB. If you catch this, you'll catch all exceptions related to DynamoDB, but no other SDK exceptions.

There are more specific exception types under `DynamoDbException`. Some of these exception types apply to control-plane operations such as `TableAlreadyExistsException`. Others apply to data-plane operations. The following is an example of a common data-plane exception:

- `ConditionalCheckFailedException` – You specified a condition in the request that evaluated to false. For example, you might have tried to perform a conditional update on an item, but the actual value of the attribute did not match the expected value in the condition. A request that fails in this manner isn't retried.

Other situations don't have a specific exception defined. For example, when your requests get throttled the specific `ProvisionedThroughputExceededException` might get thrown, while in

other cases the more generic `DynamoDbException` is thrown. In either case, you can determine if throttling caused the exception by checking if `isThrottlingException()` returns `true`.

Depending on your application needs, you can catch all `AwsServiceException` or `DynamoDbException` instances. However, you often need different behavior in different situations. The logic to deal with a condition check failure is different than that to handle throttling. Define which exceptional paths you want to deal with and make sure to test the alternative paths. This helps you make sure that you can deal with all relevant scenarios.

For lists of common errors that you might encounter, see Error handling with DynamoDB. Also see Common Errors in the *Amazon DynamoDB API Reference*. The API Reference also provides the exact errors possible for each API operation, such as for the `Query` operation. For information about handling exceptions, see Exception handling for the AWS SDK for Java 2.x in the *AWS SDK for Java 2.x Developer Guide*.

AWS request ID

Each request includes a request ID, which can be useful to pull if you're working with AWS Support to diagnose an issue. Each exception derived from `SdkServiceException` has a `requestId()` method available to retrieve the request ID.

Logging

Using the logging provided that the SDK provides can be useful both for catching any important messages from the client libraries and for more in-depth debugging purposes. Loggers are hierarchical and the SDK uses `software.amazon.awssdk` as its root logger. You can configure the level with one of `TRACE`, `DEBUG`, `INFO`, `WARN`, `ERROR`, `ALL`, or `OFF`. The configured level applies to that logger and down into the logger hierarchy.

For its logging, the AWS SDK for Java 2.x uses the Simple Logging Façade for Java (SLF4J). This acts as an abstraction layer around other loggers, and you can use it to plug in the logger that you prefer. For instructions about plugging in loggers, see the SLF4J user manual.

Each logger has a particular behavior. By default, the Log4j 2.x logger creates a `ConsoleAppender`, which appends log events to `System.out` and defaults to the `ERROR` log level.

The SimpleLogger logger included in SLF4J outputs by default to `System.err` and defaults to the `INFO` log level.

We recommend that you set the level to `WARN` for `software.amazon.awssdk` for any production deployments to catch any important messages from the SDK's client libraries while limiting the output quantity.

If SLF4J can't find a supported logger on the class path (no SLF4J binding), then it defaults to a no operation implementation. This implementation results in logging messages to `System.err` explaining that SLF4J could not find a logger implementation on the classpath. To prevent this situation, you must add a logger implementation. To do this, you can add a dependency in your Apache Maven `pom.xml` on artifacts, such as `org.slf4j.slf4j-simple` or `org.apache.logging.log4j.log4j-slf4j2-imp`.

For information about how to configure the logging in the SDK, including adding logging dependencies to your application configuration, see Logging with the SDK for Java 2.x in the *AWS SDK for Java Developer Guide*.

The following configuration in the `Log4j2.xml` file shows how to adjust the logging behavior if you use the Apache Log4j 2 logger. This configuration sets the root logger level to `WARN`. All loggers in the hierarchy inherit this log level, including the `software.amazon.awssdk` logger.

By default, the output goes to `System.out`. In the following example, we still override the default output Log4j appender to apply a tailored Log4j `PatternLayout`.

Example of a `Log4j2.xml` configuration file

The following configuration logs messages to the console at the `ERROR` and `WARN` levels for all logger hierarchies.

```
<Configuration status="WARN">
  <Appenders>
    <Console name="ConsoleAppender" target="SYSTEM_OUT">
      <PatternLayout pattern="%d{YYYY-MM-dd HH:mm:ss} [%t] %-5p %c:%L - %m%n" />
    </Console>
  </Appenders>

  <Loggers>
    <Root level="WARN">
      <AppenderRef ref="ConsoleAppender"/>
    </Root>
  </Loggers>
</Configuration>
```

AWS request ID logging

When something goes wrong, you can find request IDs within exceptions. However, if you want the request IDs for requests that aren't generating exceptions, then you can use logging.

The `software.amazon.awssdk.request` logger outputs request IDs at the DEBUG level. The following example extends the previous configuration example to keep the root logger level at ERROR, the `software.amazon.awssdk` at level WARN, and the `software.amazon.awssdk.request` at level DEBUG. Setting these levels helps to catch the request IDs and other request-related details, such as the endpoint and status code.

```
<Configuration status="WARN">
  <Appenders>
    <Console name="ConsoleAppender" target="SYSTEM_OUT">
      <PatternLayout pattern="%d{YYYY-MM-dd HH:mm:ss} [%t] %-5p %c:%L - %m%n" />
    </Console>
  </Appenders>

  <Loggers>
    <Root level="ERROR">
      <AppenderRef ref="ConsoleAppender"/>
    </Root>
    <Logger name="software.amazon.awssdk" level="WARN" />
    <Logger name="software.amazon.awssdk.request" level="DEBUG" />
  </Loggers>
</Configuration>
```

Here is an example of the log output:

```
2022-09-23 16:02:08 [main] DEBUG software.amazon.awssdk.request:85 - Sending Request:
 DefaultSdkHttpFullRequest(httpMethod=POST, protocol=https, host=dynamodb.us-
east-1.amazonaws.com, encodedPath=/, headers=[amz-sdk-invocation-id, Content-Length,
 Content-Type, User-Agent, X-Amz-Target], queryParameters=[])
 2022-09-23 16:02:08 [main] DEBUG software.amazon.awssdk.request:85 - Received
 successful response: 200, Request ID:
 QS9DUMME2NHEDH8TGT9N5V53OJVV4KQNSO5AEMVJF66Q9ASUAAJG, Extended Request ID: not
 available
```

Pagination

Some requests, such as `Query` and `Scan`, limit the size of data returned on a single request and require you make repeated requests to pull subsequent pages.

You can control the maximum number of items to read for each page with the `Limit` parameter. For example, you can use the `Limit` parameter to retrieve only the last 10 items. This limit specifies how many items to read from the table before any filtering is applied. If you want exactly 10 items after filtering, there's no way to specify that. You can control only the pre-filtered count and check client-side when you've actually retrieved 10 items. Regardless of the limit, responses always have a maximum size of 1 MB.

A `LastEvaluatedKey` might be included in the API response. This indicates that the response ended because it reached a count limit or a size limit. This key is the last key evaluated for that response. By interacting directly with the API, you can retrieve this `LastEvaluatedKey` and pass it to a follow-up call as `ExclusiveStartKey` to read the next chunk from that starting point. If no `LastEvaluatedKey` is returned, it means that there are no more items that match the `Query` or `Scan` API call.

The following example uses the low-level interface to limit the items to 100 based on the `keyConditionExpression` parameter.

```
QueryRequest.Builder queryRequestBuilder = QueryRequest.builder()
        .expressionAttributeValues(Map.of(
                ":pk_val", AttributeValue.fromS("123"),
                ":sk_val", AttributeValue.fromN("1000")))
        .keyConditionExpression("pk = :pk_val AND sk > :sk_val")
        .limit(100)
        .tableName(TABLE_NAME);

while (true) {
    QueryResponse queryResponse = DYNAMODB_CLIENT.query(queryRequestBuilder.build());

    queryResponse.items().forEach(item -> {
        LOGGER.info("item PK: [" + item.get("pk") + "] and SK: [" + item.get("sk") +
 "]");
    });

    if (!queryResponse.hasLastEvaluatedKey()) {
        break;
    }
```

```
    queryRequestBuilder.exclusiveStartKey(queryResponse.lastEvaluatedKey());
}
```

The AWS SDK for Java 2.x can simplify this interaction with DynamoDB by providing auto-pagination methods that make multiple service calls to automatically get the next pages of results for you. This simplifies your code, but it takes away some control of resource usage that you would keep by manually reading pages.

By using the `Iterable` methods available in the DynamoDB client, such as `QueryPaginator` and `ScanPaginator`, the SDK takes care of the pagination. The return type of these methods is a custom iterable that you can use to iterate through all the pages. The SDK internally handles service calls for you. Using the Java Stream API, you can handle the result of `QueryPaginator` as shown in the following example.

```
QueryPublisher queryPublisher =
    DYNAMODB_CLIENT.queryPaginator(QueryRequest.builder()
        .expressionAttributeValues(Map.of(
            ":pk_val", AttributeValue.fromS("123"),
            ":sk_val", AttributeValue.fromN("1000")))
        .keyConditionExpression("pk = :pk_val AND sk > :sk_val")
        .limit(100)
        .tableName("YourTableName")
        .build());

queryPublisher.items().subscribe(item ->
    System.out.println(item.get("itemData"))).join();
```

Data class annotations

The Java SDK provides several annotations that you can put on the attributes of your data class. These annotations influence how the SDK interacts with the attributes. By adding an annotation, you can have an attribute behave as an implicit atomic counter, maintain an auto-generated timestamp value, or track an item version number. For more information, see Data class annotations.

Error handling with DynamoDB

This section describes runtime errors and how to handle them. It also describes error messages and codes that are specific to Amazon DynamoDB. For a list of common errors that apply to all AWS services, see Access Management

Topics

- Error components
- Transactional errors
- Error messages and codes
- Error handling in your application
- Error retries and exponential backoff
- Batch operations and error handling

Error components

When your program sends a request, DynamoDB attempts to process it. If the request is successful, DynamoDB returns an HTTP success status code (`200 OK`), along with the results from the requested operation.

If the request is unsuccessful, DynamoDB returns an error. Each error has three components:

- An HTTP status code (such as `400`).
- An exception name (such as `ResourceNotFoundException`).
- An error message (such as `Requested resource not found: Table: tablename not found`).

The AWS SDKs take care of propagating errors to your application so that you can take appropriate action. For example, in a Java program, you can write `try-catch` logic to handle a `ResourceNotFoundException`.

If you are not using an AWS SDK, you need to parse the content of the low-level response from DynamoDB. The following is an example of such a response.

```
HTTP/1.1 400 Bad Request
x-amzn-RequestId: LDM6CJP8RMQ1FHKSC1RBVJFPNVV4KQNSO5AEMF66Q9ASUAAJG
Content-Type: application/x-amz-json-1.0
Content-Length: 240
Date: Thu, 15 Mar 2012 23:56:23 GMT

{"__type":"com.amazonaws.dynamodb.v20120810#ResourceNotFoundException",
"message":"Requested resource not found: Table: tablename not found"}
```

Transactional errors

For information on transactional errors, please see Transaction conflict handling in DynamoDB

Error messages and codes

The following is a list of exceptions returned by DynamoDB, grouped by HTTP status code. If *OK to retry?* is *Yes*, you can submit the same request again. If *OK to retry?* is *No*, you need to fix the problem on the client side before you submit a new request.

HTTP status code 400

An HTTP `400` status code indicates a problem with your request, such as authentication failure, missing required parameters, or exceeding a table's provisioned throughput. You have to fix the issue in your application before submitting the request again.

AccessDeniedException

Message: *Access denied.*

The client did not correctly sign the request. If you are using an AWS SDK, requests are signed for you automatically; otherwise, go to the Signature version 4 signing process in the *AWS General Reference*.

OK to retry? No

ConditionalCheckFailedException

Message: *The conditional request failed.*

You specified a condition that evaluated to false. For example, you might have tried to perform a conditional update on an item, but the actual value of the attribute did not match the expected value in the condition.

OK to retry? No

IncompleteSignatureException

Message: *The request signature does not conform to AWS standards.*

The request signature did not include all of the required components. If you are using an AWS SDK, requests are signed for you automatically; otherwise, go to the Signature Version 4 signing process in the *AWS General Reference*.

OK to retry? No

ItemCollectionSizeLimitExceededException

Message: *Collection size exceeded.*

For a table with a local secondary index, a group of items with the same partition key value has exceeded the maximum size limit of 10 GB. For more information on item collections, see Item collections in Local Secondary Indexes.

OK to retry? Yes

LimitExceededException

Message: *Too many operations for a given subscriber.*

There are too many concurrent control plane operations. The cumulative number of tables and indexes in the `CREATING`, `DELETING`, or `UPDATING` state cannot exceed 500.

OK to retry? Yes

MissingAuthenticationTokenException

Message: *Request must contain a valid (registered) AWS Access Key ID.*

The request did not include the required authorization header, or it was malformed. See DynamoDB low-level API.

OK to retry? No

ProvisionedThroughputExceededException

Message: *You exceeded your maximum allowed provisioned throughput for a table or for one or more global secondary indexes. To view performance metrics for provisioned throughput vs. consumed throughput, open the Amazon CloudWatch console.*

The error includes a list of `ThrottlingReason` fields that provides specific context about why throttling occurred, following the format `ResourceType+OperationType+LimitType` (e.g., `TableReadProvisionedThroughputExceeded`) and the ARN of the affected resource. This helps you identify which resource is being throttled (table or index), what operation type triggered the throttling (read or write), and the specific limit that was exceeded (in this case: provisioned capacity). For more information about diagnosing and resolving throttling issues, see Diagnosing throttling.

Example: Your request rate is too high. The AWS SDKs for DynamoDB automatically retry requests that receive this exception. Your request is eventually successful, unless your retry queue is too large to finish. Reduce the frequency of requests using Error retries and exponential backoff.

OK to retry? Yes

ReplicatedWriteConflictException

Message: *One or more items in this request are being modified by a request in another Region.*

Example: A write operation was requested for an item in a multi-Region strongly consistent (MRSC) global table that is currently being modified by a request in another Region.

OK to retry? Yes

RequestLimitExceeded

Message: *Throughput exceeds the current throughput limit for your account. To request a limit increase, contact AWS Support at https://aws.amazon.com/support*.

The error includes a list of `ThrottlingReason` fields that provides specific context about why throttling occurred, following the format `ResourceType+OperationType+LimitType` (e.g., `TableWriteAccountLimitExceeded` or `IndexReadAccountLimitExceeded`) and the ARN of the affected resource. This helps you identify which resource is being throttled (table or index), what operation type triggered the throttling (read or write), and that you've exceeded account-level service quotas. For more information about diagnosing and resolving throttling issues, see Diagnosing throttling.

Example: Rate of on-demand requests exceeds the allowed account throughput and the table cannot be scaled further.

OK to retry? Yes

ResourceInUseException

Message: *The resource which you are attempting to change is in use.*

Example: You tried to re-create an existing table, or delete a table currently in the `CREATING` state.

OK to retry? No

ResourceNotFoundException

Message: *Requested resource not found.*

Example: The table that is being requested does not exist, or is too early in the `CREATING` state.

OK to retry? No

ThrottlingException

Message: *Rate of requests exceeds the allowed throughput.*

This exception is returned as an AmazonServiceException response with a THROTTLING_EXCEPTION status code. This exception might be returned if you perform control plane API operations too rapidly.

For tables using on-demand mode, this exception might be returned for any data plane API operation if your request rate is too high. To learn more about on-demand scaling, see Initial throughput and scaling properties.

The error includes a list of `ThrottlingReason` fields that provides specific context about why throttling occurred, following the format `ResourceType +OperationType+LimitType` (e.g., `TableReadKeyRangeThroughputExceeded` or `IndexWriteMaxOnDemandThroughputExceeded`) and the ARN of the affected resource. This information helps you identify which resource is being throttled (table or index), what operation type triggered the throttling (read or write), and the specific limit that was exceeded (partition limits or on-demand maximum throughput). For more information about diagnosing and resolving throttling issues, see Diagnosing throttling.

OK to retry? Yes

UnrecognizedClientException

Message: *The Access Key ID or security token is invalid.*

The request signature is incorrect. The most likely cause is an invalid AWS access key ID or secret key.

OK to retry? Yes

ValidationException

Message: Varies, depending upon the specific error(s) encountered

This error can occur for several reasons, such as a required parameter that is missing, a value that is out of range, or mismatched data types. The error message contains details about the specific part of the request that caused the error.

OK to retry? No

HTTP status code 5xx

An HTTP `5xx` status code indicates a problem that must be resolved by AWS. This might be a transient error, in which case you can retry your request until it succeeds. Otherwise, go to the AWS Service Health Dashboard to see if there are any operational issues with the service.

For more information, see How do I resolve HTTP 5xx errors in Amazon DynamoDB?

InternalServerError (HTTP 500)

DynamoDB could not process your request.

OK to retry? Yes

> **Note**
>
> You might encounter internal server errors while working with items. These are expected during the lifetime of a table. Any failed requests can be retried immediately. When you receive a status code 500 on a write operation, the operation may have succeeded or failed. If the write operation is a `TransactWriteItem` request, then it is

OK to retry the operation. If the write operation is a single-item write request such as `PutItem`, `UpdateItem`, or `DeleteItem`, then your application should read the state of the item before retrying the operation, and/or use DynamoDB condition expression CLI example to ensure that the item remains in a correct state after retrying regardless of whether the prior operation succeeded or failed. If idempotency is a requirement for the write operation, please use `TransactWriteItem`, which supports idempotent requests by automatically specifying a `ClientRequestToken` to disambiguate multiple attempts to perform the same action.

ServiceUnavailable (HTTP 503)

DynamoDB is currently unavailable. (This should be a temporary state.)

OK to retry? Yes

Error handling in your application

For your application to run smoothly, you need to add logic to catch and respond to errors. Typical approaches include using `try-catch` blocks or `if-then` statements.

The AWS SDKs perform their own retries and error checking. If you encounter an error while using one of the AWS SDKs, the error code and description can help you troubleshoot it.

You should also see a `Request ID` in the response. The `Request ID` can be helpful if you need to work with AWS Support to diagnose an issue.

Error retries and exponential backoff

Numerous components on a network, such as DNS servers, switches, load balancers, and others, can generate errors anywhere in the life of a given request. The usual technique for dealing with these error responses in a networked environment is to implement retries in the client application. This technique increases the reliability of the application.

Each AWS SDK implements retry logic automatically. You can modify the retry parameters to your needs. For example, consider a Java application that requires a fail-fast strategy, with no retries allowed in case of an error. With the AWS SDK for Java, you could use the `ClientConfiguration` class and provide a `maxErrorRetry` value of `0` to turn off the retries. For more information, see the AWS SDK documentation for your programming language.

If you're not using an AWS SDK, you should retry original requests that receive server errors (5xx). However, client errors (4xx, other than a `ThrottlingException` or a `ProvisionedThroughputExceededException`) indicate that you need to revise the request itself to correct the problem before trying again. For detailed recommendations to address specific throttling scenarios, refer to the DynamoDB throttling troubleshooting section.

In addition to simple retries, each AWS SDK implements an exponential backoff algorithm for better flow control. The concept behind exponential backoff is to use progressively longer waits between retries for consecutive error responses. For example, up to 50 milliseconds before the first retry, up to 100 milliseconds before the second, up to 200 milliseconds before third, and so on. However, after a minute, if the request has not succeeded, the problem might be the request size exceeding your provisioned throughput, and not the request rate. Set the maximum number of retries to stop around one minute. If the request is not successful, investigate your provisioned throughput options.

> **Note**
>
> The AWS SDKs implement automatic retry logic and exponential backoff.

Most exponential backoff algorithms use jitter (randomized delay) to prevent successive collisions. Because you aren't trying to avoid such collisions in these cases, you do not need to use this random number. However, if you use concurrent clients, jitter can help your requests succeed faster. For more information, see the blog post about Exponential backoff and jitter.

Batch operations and error handling

The DynamoDB low-level API supports batch operations for reads and writes. `BatchGetItem` reads items from one or more tables, and `BatchWriteItem` puts or deletes items in one or more tables. These batch operations are implemented as wrappers around other non-batch DynamoDB operations. In other words, `BatchGetItem` invokes `GetItem` once for each item in the batch. Similarly,`BatchWriteItem` invokes `DeleteItem` or `PutItem`, as appropriate, for each item in the batch.

A batch operation can tolerate the failure of individual requests in the batch. For example, consider a `BatchGetItem` request to read five items. Even if some of the underlying `GetItem` requests fail, this does not cause the entire `BatchGetItem` operation to fail. However, if all five read operations fail, then the entire `BatchGetItem` fails.

The batch operations return information about individual requests that fail so that you can diagnose the problem and retry the operation. For `BatchGetItem`, the tables and primary keys in question are returned in the `UnprocessedKeys` value of the response. For `BatchWriteItem`, similar information is returned in `UnprocessedItems`.

The most likely cause of a failed read or a failed write is *throttling*. For `BatchGetItem`, one or more of the tables in the batch request does not have enough provisioned read capacity to support the operation. For `BatchWriteItem`, one or more of the tables does not have enough provisioned write capacity.

If DynamoDB returns any unprocessed items, you should retry the batch operation on those items. However, *we strongly recommend that you use an exponential backoff algorithm*. If you retry the batch operation immediately, the underlying read or write requests can still fail due to throttling on the individual tables. If you delay the batch operation using exponential backoff, the individual requests in the batch are much more likely to succeed.

Note

Some AWS SDKs provide higher-level clients that handle unprocessed item retries automatically, so you don't need to implement this retry logic yourself. For example:

- **Java** – The DynamoDB Enhanced Client in the AWS SDK for Java v2 and the DynamoDBMapper in v1 both automatically retry unprocessed items when performing batch operations.
- **Python** – The boto3 Table resource `batch_writer` handles unprocessed item retries implicitly for batch write operations. For more information, see the section called "Using batch_writer".

If you are using a low-level client or an SDK that does not provide this behavior, you must implement the retry logic yourself as described above.

Using DynamoDB with an AWS SDK

AWS software development kits (SDKs) are available for many popular programming languages. Each SDK provides an API, code examples, and documentation that make it easier for developers to build applications in their preferred language.

SDK documentation	Code examples
AWS SDK for C++	AWS SDK for C++ code examples
AWS CLI	AWS CLI code examples
AWS SDK for Go	AWS SDK for Go code examples
AWS SDK for Java	AWS SDK for Java code examples
AWS SDK for JavaScript	AWS SDK for JavaScript code examples
AWS SDK for Kotlin	AWS SDK for Kotlin code examples
AWS SDK for .NET	AWS SDK for .NET code examples
AWS SDK for PHP	AWS SDK for PHP code examples
AWS Tools for PowerShell	AWS Tools for PowerShell code examples
AWS SDK for Python (Boto3)	AWS SDK for Python (Boto3) code examples
AWS SDK for Ruby	AWS SDK for Ruby code examples
AWS SDK for Rust	AWS SDK for Rust code examples
AWS SDK for SAP ABAP	AWS SDK for SAP ABAP code examples
AWS SDK for Swift	AWS SDK for Swift code examples

For examples specific to DynamoDB, see Code examples for DynamoDB using AWS SDKs.

Example availability

Can't find what you need? Request a code example by using the **Provide feedback** link at the bottom of this page.

Working with tables, items, queries, scans, and indexes

This section provides details about working with tables, items, queries, and more in Amazon DynamoDB.

Topics

- Working with tables and data in DynamoDB
- Global tables - multi-active, multi-Region replication
- Working with items and attributes in DynamoDB
- Improving data access with secondary indexes in DynamoDB
- Managing complex workflows with DynamoDB transactions
- Change data capture with Amazon DynamoDB

Working with tables and data in DynamoDB

This section describes how to use the AWS Command Line Interface (AWS CLI) and the AWS SDKs to create, update, and delete tables in Amazon DynamoDB.

> **Note**
>
> You can also perform these same tasks using the AWS Management Console. For more information, see Using the console.

This section also provides more information about throughput capacity using DynamoDB auto scaling or manually setting provisioned throughput.

Topics

- Basic operations on DynamoDB tables
- Considerations when choosing a table class in DynamoDB
- Adding tags and labels to resources in DynamoDB

Basic operations on DynamoDB tables

Similar to other database systems, Amazon DynamoDB stores data in tables. You can manage your tables using a few basic operations.

Topics

- Creating a table
- Describing a table
- Updating a table
- Deleting a table
- Using deletion protection
- Listing table names
- Describing provisioned throughput quotas

Creating a table

Use the `CreateTable` operation to create a table in Amazon DynamoDB. To create the table, you must provide the following information:

- **Table name.** The name must conform to the DynamoDB naming rules, and must be unique for the current AWS account and Region. For example, you could create a `People` table in US East (N. Virginia) and another `People` table in Europe (Ireland). However, these two tables would be entirely different from each other. For more information, see Supported data types and naming rules in Amazon DynamoDB.
- **Primary key.** The primary key can consist of one attribute (partition key) or two attributes (partition key and sort key). You need to provide the attribute names, data types, and the role of each attribute: `HASH` (for a partition key) and `RANGE` (for a sort key). For more information, see Primary key.
- **Throughput settings (for provisioned tables).** If using provisioned mode, you must specify the initial read and write throughput settings for the table. You can modify these settings later, or enable DynamoDB auto scaling to manage the settings for you. For more information, see DynamoDB provisioned capacity mode and Managing throughput capacity automatically with DynamoDB auto scaling.

Example 1: Create an on-demand table

To create the same table `Music` using on-demand mode.

```
aws dynamodb create-table \
    --table-name Music \
    --attribute-definitions \
        AttributeName=Artist,AttributeType=S \
        AttributeName=SongTitle,AttributeType=S \
    --key-schema \
        AttributeName=Artist,KeyType=HASH \
        AttributeName=SongTitle,KeyType=RANGE \
    --billing-mode=PAY_PER_REQUEST
```

The `CreateTable` operation returns metadata for the table, as shown following.

```
{
    "TableDescription": {
        "TableArn": "arn:aws:dynamodb:us-east-1:123456789012:table/Music",
        "AttributeDefinitions": [
            {
                "AttributeName": "Artist",
                "AttributeType": "S"
            },
            {
                "AttributeName": "SongTitle",
                "AttributeType": "S"
            }
        ],
        "ProvisionedThroughput": {
            "NumberOfDecreasesToday": 0,
            "WriteCapacityUnits": 0,
            "ReadCapacityUnits": 0
        },
        "TableSizeBytes": 0,
        "TableName": "Music",
        "BillingModeSummary": {
            "BillingMode": "PAY_PER_REQUEST"
        },
        "TableStatus": "CREATING",
        "TableId": "12345678-0123-4567-a123-abcdefghijkl",
        "KeySchema": [
            {
```

```
                "KeyType": "HASH",
                "AttributeName": "Artist"
            },
            {
                "KeyType": "RANGE",
                "AttributeName": "SongTitle"
            }
        ],
        "ItemCount": 0,
        "CreationDateTime": 1542397468.348
    }
}
```

Important

When calling `DescribeTable` on an on-demand table, read capacity units and write capacity units are set to 0.

Example 2: Create a provisioned table

The following AWS CLI example shows how to create a table (`Music`). The primary key consists of `Artist` (partition key) and `SongTitle` (sort key), each of which has a data type of `String`. The maximum throughput for this table is 10 read capacity units and 5 write capacity units.

```
aws dynamodb create-table \
    --table-name Music \
    --attribute-definitions \
        AttributeName=Artist,AttributeType=S \
        AttributeName=SongTitle,AttributeType=S \
    --key-schema \
        AttributeName=Artist,KeyType=HASH \
        AttributeName=SongTitle,KeyType=RANGE \
    --provisioned-throughput \
        ReadCapacityUnits=10,WriteCapacityUnits=5
```

The `CreateTable` operation returns metadata for the table, as shown following.

```
{
    "TableDescription": {
        "TableArn": "arn:aws:dynamodb:us-east-1:123456789012:table/Music",
        "AttributeDefinitions": [
```

```
            {
                "AttributeName": "Artist",
                "AttributeType": "S"
            },
            {
                "AttributeName": "SongTitle",
                "AttributeType": "S"
            }
        ],
        "ProvisionedThroughput": {
            "NumberOfDecreasesToday": 0,
            "WriteCapacityUnits": 5,
            "ReadCapacityUnits": 10
        },
        "TableSizeBytes": 0,
        "TableName": "Music",
        "TableStatus": "CREATING",
        "TableId": "12345678-0123-4567-a123-abcdefghijkl",
        "KeySchema": [
            {
                "KeyType": "HASH",
                "AttributeName": "Artist"
            },
            {
                "KeyType": "RANGE",
                "AttributeName": "SongTitle"
            }
        ],
        "ItemCount": 0,
        "CreationDateTime": 1542397215.37
    }
}
```

The `TableStatus` element indicates the current state of the table (`CREATING`). It might take a while to create the table, depending on the values you specify for `ReadCapacityUnits` and `WriteCapacityUnits`. Larger values for these require DynamoDB to allocate more resources for the table.

Example 3: Create a table using the DynamoDB standard-infrequent access table class

To create the same `Music` table using the DynamoDB Standard-Infrequent Access table class.

```
aws dynamodb create-table \
```

```
    --table-name Music \
    --attribute-definitions \
        AttributeName=Artist,AttributeType=S \
        AttributeName=SongTitle,AttributeType=S \
    --key-schema \
        AttributeName=Artist,KeyType=HASH \
        AttributeName=SongTitle,KeyType=RANGE \
    --provisioned-throughput \
        ReadCapacityUnits=10,WriteCapacityUnits=5 \
    --table-class STANDARD_INFREQUENT_ACCESS
```

The `CreateTable` operation returns metadata for the table, as shown following.

```
{
    "TableDescription": {
        "TableArn": "arn:aws:dynamodb:us-east-1:123456789012:table/Music",
        "AttributeDefinitions": [
            {
                "AttributeName": "Artist",
                "AttributeType": "S"
            },
            {
                "AttributeName": "SongTitle",
                "AttributeType": "S"
            }
        ],
        "ProvisionedThroughput": {
            "NumberOfDecreasesToday": 0,
            "WriteCapacityUnits": 5,
            "ReadCapacityUnits": 10
        },
        "TableClassSummary": {
            "LastUpdateDateTime": 1542397215.37,
            "TableClass": "STANDARD_INFREQUENT_ACCESS"
        },
        "TableSizeBytes": 0,
        "TableName": "Music",
        "TableStatus": "CREATING",
        "TableId": "12345678-0123-4567-a123-abcdefghijkl",
        "KeySchema": [
            {
                "KeyType": "HASH",
                "AttributeName": "Artist"
```

```
            },
            {
                "KeyType": "RANGE",
                "AttributeName": "SongTitle"
            }
        ],
        "ItemCount": 0,
        "CreationDateTime": 1542397215.37
    }
}
```

Describing a table

To view details about a table, use the `DescribeTable` operation. You must provide the table name. The output from `DescribeTable` is in the same format as that from `CreateTable`. It includes the timestamp when the table was created, its key schema, its provisioned throughput settings, its estimated size, and any secondary indexes that are present.

> **Important**
>
> When calling `DescribeTable` on an on-demand table, read capacity units and write capacity units are set to 0.

Example

```
aws dynamodb describe-table --table-name Music
```

The table is ready for use when the `TableStatus` has changed from `CREATING` to `ACTIVE`.

> **Note**
>
> If you issue a `DescribeTable` request immediately after a `CreateTable` request, DynamoDB might return an error (`ResourceNotFoundException`). This is because `DescribeTable` uses an eventually consistent query, and the metadata for your table might not be available at that moment. Wait for a few seconds, and then try the `DescribeTable` request again.

For billing purposes, your DynamoDB storage costs include a per-item overhead of 100 bytes. (For more information, go to DynamoDB Pricing.) This extra 100 bytes per item is not used in capacity unit calculations or by the `DescribeTable` operation.

Updating a table

The `UpdateTable` operation allows you to do one of the following:

- Modify a table's provisioned throughput settings (for provisioned mode tables).
- Change the table's read/write capacity mode.
- Manipulate global secondary indexes on the table (see Using Global Secondary Indexes in DynamoDB).
- Enable or disable DynamoDB Streams on the table (see Change data capture for DynamoDB Streams).

Example

The following AWS CLI example shows how to modify a table's provisioned throughput settings.

```
aws dynamodb update-table --table-name Music \
    --provisioned-throughput ReadCapacityUnits=20,WriteCapacityUnits=10
```

Note

When you issue an `UpdateTable` request, the status of the table changes from `AVAILABLE` to `UPDATING`. The table remains fully available for use while it is `UPDATING`. When this process is completed, the table status changes from `UPDATING` to `AVAILABLE`.

Example

The following AWS CLI example shows how to modify a table's read/write capacity mode to on-demand mode.

```
aws dynamodb update-table --table-name Music \
    --billing-mode PAY_PER_REQUEST
```

Deleting a table

You can remove an unused table with the `DeleteTable` operation. Deleting a table is an unrecoverable operation. To delete a table using the AWS Management Console, see the section called “Step 6: (Optional) clean up”.

Example

The following AWS CLI example shows how to delete a table.

```
aws dynamodb delete-table --table-name Music
```

When you issue a `DeleteTable` request, the table's status changes from `ACTIVE` to `DELETING`. It might take a while to delete the table, depending on the resources it uses (such as the data stored in the table, and any streams or indexes on the table).

When the `DeleteTable` operation concludes, the table no longer exists in DynamoDB.

Using deletion protection

You can protect a table from accidental deletion with the deletion protection property. Enabling this property for tables helps ensure that tables do not get accidentally deleted during regular table management operations by your administrators. This will help prevent disruption to your normal business operations.

The table owner or an authorized administrator controls the deletion protection property for each table. The deletion protection property for every table is off by default. This includes global replicas, and tables restored from backups. When deletion protection is disabled for a table, the table can be deleted by any users authorized by an Identity and Access Management (IAM) policy. When deletion protection is enabled for a table, it cannot be deleted by anyone.

To change this setting, go to the table’s **Additional settings**, navigate to the **Deletion Protection** panel and select **Enable delete protection**.

The deletion protection property is supported by the DynamoDB console, API, CLI/SDK and CloudFormation. The `CreateTable` API supports the deletion protection property at table creation time, and the `UpdateTable` API supports changing the deletion protection property for existing tables.

Note

- If an AWS account is deleted, all of that account's data including tables are still deleted within 90 days.
- If DynamoDB loses access to a customer managed key that was used to encrypt a table, it will still archive the table. Archiving involves making a backup of the table and deleting the original.

Listing table names

The `ListTables` operation returns the names of the DynamoDB tables for the current AWS account and Region.

Example

The following AWS CLI example shows how to list the DynamoDB table names.

```
aws dynamodb list-tables
```

Describing provisioned throughput quotas

The `DescribeLimits` operation returns the current read and write capacity quotas for the current AWS account and Region.

Example

The following AWS CLI example shows how to describe the current provisioned throughput quotas.

```
aws dynamodb describe-limits
```

The output shows the upper quotas of read and write capacity units for the current AWS account and Region.

For more information about these quotas, and how to request quota increases, see Throughput default quotas.

Considerations when choosing a table class in DynamoDB

DynamoDB offers two table classes designed to help you optimize for cost. The DynamoDB Standard table class is the default, and is recommended for the vast majority of workloads. The DynamoDB Standard-Infrequent Access (DynamoDB Standard-IA) table class is optimized for tables where storage is the dominant cost. For example, tables that store infrequently accessed data, such as application logs, old social media posts, e-commerce order history, and past gaming achievements, are good candidates for the Standard-IA table class.

Every DynamoDB table is associated with a table class. All secondary indexes associated with the table use the same table class. You can set your table class when creating your table (DynamoDB Standard by default) and update the table class of an existing table using the AWS Management Console, AWS CLI, or AWS SDK. DynamoDB also supports managing your table class using AWS CloudFormation for single-region tables (tables that are not global tables). Each table class offers different pricing for data storage as well as read and write requests. When choosing a table class for your table, keep the following in mind:

- The DynamoDB Standard table class offers lower throughput costs than DynamoDB Standard-IA and is the most cost-effective option for tables where throughput is the dominant cost.
- The DynamoDB Standard-IA table class offers lower storage costs than DynamoDB Standard, and is the most cost-effective option for tables where storage is the dominant cost. When storage exceeds 50% of the throughput (reads and writes) cost of a table using the DynamoDB Standard table class, the DynamoDB Standard-IA table class can help you reduce your total table cost.
- DynamoDB Standard-IA tables offer the same performance, durability, and availability as DynamoDB Standard tables.
- Switching between the DynamoDB Standard and DynamoDB Standard-IA table classes does not require changing your application code. You use the same DynamoDB APIs and service endpoints regardless of the table class your tables use.
- DynamoDB Standard-IA tables are compatible with all existing DynamoDB features such as auto scaling, on-demand mode, time-to-live (TTL), on-demand backups, point-in-time recovery (PITR), and global secondary indexes.

The most cost-effective table class for your table depends on your table's expected storage and throughput usage patterns. You can look at your table's historical storage and throughput cost and usage with AWS Cost and Usage Reports and the AWS Cost Explorer. Use this historical data to determine the most cost-effective table class for your table. To learn more about using AWS

Cost and Usage Reports and the AWS Cost Explorer, see the AWS Billing and Cost Management Documentation. See Amazon DynamoDB Pricing for details about table class pricing.

> **Note**
>
> A table class update is a background process. You can still access your table normally during a table class update. The time to update your table class depends on your table traffic, storage size, and other related variables. No more than two table class updates on your table are allowed in a 30-day trailing period.

Adding tags and labels to resources in DynamoDB

You can label Amazon DynamoDB resources using *tags*. Tags let you categorize your resources in different ways, for example, by purpose, owner, environment, or other criteria. Tags can help you do the following:

- Quickly identify a resource based on the tags that you assigned to it.
- See AWS bills broken down by tags.

> **Note**
>
> Any local secondary indexes (LSI) and global secondary indexes (GSI) related to tagged tables are labeled with the same tags automatically. Currently, DynamoDB Streams usage cannot be tagged.

Tagging is supported by AWS services like Amazon EC2, Amazon S3, DynamoDB, and more. Efficient tagging can provide cost insights by enabling you to create reports across services that carry a specific tag.

To get started with tagging, do the following:

1. Understand Tagging restrictions in DynamoDB.
2. Create tags by using Tagging resources in DynamoDB.
3. Use Using DynamoDB tags to create cost allocation reports to track your AWS costs per active tag.

Finally, it is good practice to follow optimal tagging strategies. For information, see AWS tagging strategies.

Tagging restrictions in DynamoDB

Each tag consists of a key and a value, both of which you define. The following restrictions apply:

- Each DynamoDB table can have only one tag with the same key. If you try to add an existing tag (same key), the existing tag value is updated to the new value.
- Tag keys and values are case sensitive.
- The maximum key length is 128 Unicode characters.
- The maximum value length is 256 Unicode characters.
- The allowed characters are letters, white space, and numbers, plus the following special characters: `+ - = . _ : /`
- The maximum number of tags per resource is 50.
- The maximum size supported for all the tags in a table is 10 KB.
- AWS-assigned tag names and values are automatically assigned the `aws:` prefix, which you can't assign. AWS-assigned tag names don't count toward the tag limit of 50 or the 10 KB maximum size limit. User-assigned tag names have the prefix `user:` in the cost allocation report.
- You can't backdate the application of a tag.

Tagging resources in DynamoDB

You can use the Amazon DynamoDB console or the AWS Command Line Interface (AWS CLI) to add, list, edit, or delete tags. You can then activate these user-defined tags so that they appear on the AWS Billing and Cost Management console for cost allocation tracking. For more information, see Using DynamoDB tags to create cost allocation reports.

For bulk editing, you can also use Tag Editor on the AWS Management Console. For more information, see Working with Tag Editor.

To use the DynamoDB API instead, see the following operations in the Amazon DynamoDB API Reference:

- TagResource
- UntagResource

- ListTagsOfResource

Topics

- Setting permissions to filter by tags
- Adding tags to new or existing tables (AWS Management Console)
- Adding tags to new or existing tables (AWS CLI)

Setting permissions to filter by tags

To use tags to filter your table list in the DynamoDB console, make sure your user's policies include access to the following operations:

- `tag:GetTagKeys`
- `tag:GetTagValues`

You can access these operations by attaching a new IAM policy to your user by following the steps below.

1. Go to the IAM console with an Admin user.
2. Select "Policies" in the left navigation menu.
3. Select "Create policy."
4. Paste the following policy into the JSON editor.

   JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "tag:GetTagKeys",
                "tag:GetTagValues"
            ],
            "Resource": "*"
        }
```

```
    ]
}
```

5. Complete the wizard and assign a name to the policy (for example, `TagKeysAndValuesReadAccess`).
6. Select "Users" in the left navigation menu.
7. From the list, select the user you normally use to access the DynamoDB console.
8. Select "Add permissions."
9. Select "Attach existing policies directly."
10. From the list, select the policy you created previously.
11. Complete the wizard.

Adding tags to new or existing tables (AWS Management Console)

You can use the DynamoDB console to add tags to new tables when you create them, or to add, edit, or delete tags for existing tables.

To tag resources on creation (console)

1. Sign in to the AWS Management Console and open the DynamoDB console at https://console.aws.amazon.com/dynamodb/.
2. In the navigation pane, choose **Tables**, and then choose **Create table**.
3. On the **Create DynamoDB table** page, provide a name and primary key. In the **Tags** section, choose **Add new tag** and enter the tags that you want to use.

   For information about tag structure, see Tagging restrictions in DynamoDB.

   For more information about creating tables, see Basic operations on DynamoDB tables.

To tag existing resources (console)

Open the DynamoDB console at https://console.aws.amazon.com/dynamodb/.

1. In the navigation pane, choose **Tables**.
2. Choose a table in the list, and then choose the **Additional settings** tab. You can add, edit, or delete your tags in the **Tags** section at the bottom of the page.

Adding tags to new or existing tables (AWS CLI)

The following examples show how to use the AWS CLI to specify tags when you create tables and indexes, and to tag existing resources.

To tag resources on creation (AWS CLI)

- The following example creates a new `Movies` table and adds the `Owner` tag with a value of `blueTeam`:

```
aws dynamodb create-table \
    --table-name Movies \
    --attribute-definitions AttributeName=Title,AttributeType=S \
    --key-schema AttributeName=Title,KeyType=HASH \
    --provisioned-throughput ReadCapacityUnits=5,WriteCapacityUnits=5 \
    --tags Key=Owner,Value=blueTeam
```

To tag existing resources (AWS CLI)

- The following example adds the `Owner` tag with a value of `blueTeam` for the `Movies` table:

```
aws dynamodb tag-resource \
    --resource-arn arn:aws:dynamodb:us-east-1:123456789012:table/Movies \
    --tags Key=Owner,Value=blueTeam
```

To list all tags for a table (AWS CLI)

- The following example lists all the tags that are associated with the `Movies` table:

```
aws dynamodb list-tags-of-resource \
    --resource-arn arn:aws:dynamodb:us-east-1:123456789012:table/Movies
```

Using DynamoDB tags to create cost allocation reports

AWS uses tags to organize resource costs on your cost allocation report. AWS provides two types of cost allocation tags:

- An AWS-generated tag. AWS defines, creates, and applies this tag for you.

- User-defined tags. You define, create, and apply these tags.

You must activate both types of tags separately before they can appear in Cost Explorer or on a cost allocation report.

To activate AWS-generated tags:

1. Sign in to the AWS Management Console and open the Billing and Cost Management console at https://console.aws.amazon.com/billing/home#/.
2. In the navigation pane, choose **Cost Allocation Tags**.
3. Under **AWS-Generated Cost Allocation Tags**, choose **Activate**.

To activate user-defined tags:

1. Sign in to the AWS Management Console and open the Billing and Cost Management console at https://console.aws.amazon.com/billing/home#/.
2. In the navigation pane, choose **Cost Allocation Tags**.
3. Under **User-Defined Cost Allocation Tags**, choose **Activate**.

After you create and activate tags, AWS generates a cost allocation report with your usage and costs grouped by your active tags. The cost allocation report includes all of your AWS costs for each billing period. The report includes both tagged and untagged resources, so that you can clearly organize the charges for resources.

> **Note**
>
> Currently, any data transferred out from DynamoDB won't be broken down by tags on cost allocation reports.

For more information, see Using cost allocation tags.

Global tables - multi-active, multi-Region replication

Amazon DynamoDB global tables is a fully managed, multi-Region, and multi-active database feature that provides easy to use data replication and fast local read and write performance for globally scaled applications.

Global tables automatically replicate your DynamoDB table data across AWS Regions and optionally across AWS accounts without requiring you to build and maintain your own replication solution. Global tables are ideal for applications requiring business continuity and high availability through multi-Region deployment. Any global table replica can serve reads and writes. Applications can achieve high resilience with a low or zero Recovery Point Objective (RPO) by shifting traffic to a different Region if application processing is interrupted in a Region. Global tables are available in all Regions where DynamoDB is available.

Consistency modes

When you create a global table, you can configure its consistency mode. Global tables support two consistency modes: multi-Region eventual consistency (MREC) and multi-Region strong consistency (MRSC).

If you do not specify a consistency mode when creating a global table, the global table defaults to multi-Region eventual consistency (MREC). A global table cannot contain replicas configured with different consistency modes. You cannot change a global table's consistency mode after creation.

Account configurations

DynamoDB now supports two global tables models, each designed for different architectural patterns:

- **Same-account global tables** – All replicas are created and managed within a single AWS account.
- **Multi-account global tables** – Replicas are deployed across multiple AWS accounts while participating in a shared replication group.

Both same-account and multi-account models support multi-Region writes, asynchronous replication, last-writer-wins conflict resolution, and the same billing model. However, they differ in how accounts, permissions, encryption, and table governance are managed.

Global tables configured for MRSC only support same-account configurations.

You can configure a global table using the AWS Management Console. Global tables use existing DynamoDB APIs to read and write data to your tables, so no application changes are required. You pay only for the resources you provision or use, with no upfront costs or commitments.

Comparison of same-account and multi-account global tables

Properties	Same-Account global tables	Multi-account global tables
Primary use case	Multi-Region resiliency for applications within a single AWS account	Multi-Region, multi-account replication for applications owned by different teams, distinct business units, or strong security boundaries across accounts
Account model	All replicas created and managed in one AWS account	Replicas created across multiple AWS accounts within the same deployment
Resource ownership	A single account owns the table and all replicas	Each account owns its local replica; replication group spans accounts
Version supported	Global tables version 2019.11.21 (Current) and Version 2017.11.29 (Legacy)	Global tables version 2019.11.21 (Current)
Control plane operations	Create, modify, and delete replicas through the table owner account	Distributed control-plane operations: accounts join or leave the replication group
Data plane operations	Standard DynamoDB endpoints per Region	Data-plane access per account/Region; routing through replication group
Security boundary	A single IAM and KMS boundary	Distinct IAM, KMS, billing, CloudTrail, and governance per account
Best fit	Organizations with centraliz ed ownership of tables	Organizations with federated teams, governance boundarie s, or multi-account setups

Topics

- Global tables core concepts
- DynamoDB same-account global table
- DynamoDB multi-account global tables
- Understanding Amazon DynamoDB billing for global tables
- DynamoDB global tables versions
- Best practices for global tables

Global tables core concepts

The following sections describe the concepts and behaviors of global tables in Amazon DynamoDB

Concepts

Global tables is a DynamoDB feature that replicates table data across AWS Regions.

A *replica table* (or replica) is a DynamoDB table that functions as part of a global table. A global table consists of two or more replica tables across different AWS Regions. Each global table can have only one replica per AWS Region. All replicas in a global table share the same table name, primary key schema, and item data.

When an application writes data to a replica in one Region, DynamoDB automatically replicates the write to all other replicas in the global table. For more information about how to get started with global tables, see Tutorials: Creating global tables or Tutorials: Creating multi-account global tables.

Versions

There are two versions of DynamoDB global tables available: Global Tables version 2019.11.21 (Current) and Global tables version 2017.11.29 (Legacy). You should use Global Tables version 2019.11.21 (Current) whenever possible. The information in this documentation section is for Version 2019.11.21 (Current). For more information, see Determining the version of a global table Determining the version of a global table.

Availability

Global tables help improve your business continuity by making it easier to implement a multi-Region high availability architecture. If a workload in a single AWS Region becomes impaired, you

can shift application traffic to a different Region and perform reads and writes to a different replica table in the same global table.

Each replica table in a global table provides the same durability and availability as a single-Region DynamoDB table. Global tables offer a 99.999% availability Service Level Agreement (SLA), compared to 99.99% for single-Region tables.

Fault injection testing

Both MREC and MRSC global tables integrate with AWS Fault Injection Service (AWS FIS), a fully managed service for running controlled fault injection experiments to improve an application's resilience. Using AWS FIS, you can:

- Create experiment templates that define specific failure scenarios.
- Inject failures to validate application resilience by simulating Region isolation (that is, pausing replication to and from a selected replica) to test error handling, recovery mechanisms, and multi-Region traffic shift behavior when one AWS Region experiences disruption.

For example, in a global table with replicas in US East (N. Virginia), US East (Ohio), and US West (Oregon), you can run an experiment in US East (Ohio) to test region isolation there while US East (N. Virginia) and US West (Oregon) continue normal operations. This controlled testing helps you identify and resolve potential issues before they affect production workloads.

See Action targets in the *AWS FIS user guide* for a complete list of AWS FIS supported actions and Cross-Region Connectivity to pause DynamoDB replication between regions.

For information about Amazon DynamoDB global table actions available in AWS FIS, see DynamoDB global tables actions reference in the *AWS FIS User Guide*.

To get started running fault injection experiments, see Planning your AWS FIS experiments in the AWS FIS user guide.

> **Note**
>
> During AWS FIS experiments in MRSC, eventually consistent reads are permitted, but table setting updates - such as changing billing mode or configuring table throughput - are not allowed, similar to MREC. Please check the CloudWatch metric `FaultInjectionServiceInducedErrors` for additional details regarding the error code.

Time To Live (TTL)

Global tables configured for MREC support configuring Time To Live (TTL) deletion. TTL settings are automatically synchronized for all replicas in a global table. When TTL deletes an item from a replica in a Region, the delete is replicated to all other replicas in the global table. TTL does not consume write capacity, so you are not charged for the TTL delete in the Region where the delete occurred. However, you are charged for the replicated delete in each other region with a replica in the global table.

TTL delete replication consumes write capacity on the replicas to which the delete is being replicated. Replicas configured for provisioned capacity may throttle requests if the combination of write throughput and TTL delete throughput is higher than the provisioned write capacity.

Global tables configured for multi-Region strong consistency (MRSC) do not support configuring Time To Live (TTL) deletion.

Streams

Global tables configured for multi-Region eventual consistency (MREC) replicate changes by reading those changes from a DynamoDB Stream on a replica table and applying that change to all other replica tables. Streams are therefore enabled by default on all replicas in an MREC global table, and cannot be disabled on those replicas. The MREC replication process may combine multiple changes in a short period of time into a single replicated write, resulting in each replica's Stream containing slightly different records. Streams records on MREC replicas maintain ordering for all changes to the same item, but the relative ordering of changes to different items may vary across replicas.

If you want to write an application that processes Streams records for changes that occurred in a particular Region but not other Regions in a global table, you can add an attribute to each item that defines in which Region the change for that item occurred. You can use this attribute to filter Streams records for changes that occurred in other Regions, including the use of Lambda event filters to only invoke Lambda functions for changes in a specific Region.

Global tables configured for multi-Region strong consistency (MRSC) do not use DynamoDB Streams for replication, so Streams are not enabled by default on MRSC replicas. You can enable Streams on an MRSC replica. Streams records on MRSC replicas are identical for every replica, including Stream record ordering.

Transactions

On a global table configured for MREC, DynamoDB transaction operations (`TransactWriteItems` and `TransactGetItems`) are only atomic within the Region where the operation was invoked. Transactional writes are not replicated as a unit across Regions, meaning only some of the writes in a transaction may be returned by read operations in other replicas at a given point in time.

For example, if you have a global table with replicas in the US East (Ohio) and US West (Oregon) Regions and perform a `TransactWriteItems` operation in the US East (Ohio) Region, you may observe partially completed transactions in the US West (Oregon) Region as changes are replicated. Changes will only be replicated to other Regions once they've been committed in the source Region.

Global tables configured for multi-Region strong consistency (MRSC) do not support transaction operations, and will return an error if those operations are invoked on an MRSC replica.

Read and write throughput

Provisioned mode

Replication consumes write capacity. Replicas configured for provisioned capacity may throttle requests if the combination of application write throughput and replication write throughput exceeds the provisioned write capacity. For global tables using provisioned mode, auto scaling settings for both read and write capacities are synchronized between replicas.

You can independently configure read capacity settings for each replica in a global table by using the `ProvisionedThroughputOverride` parameter at the replica level. By default, changes to provisioned read capacity are applied to all replicas in the global table. When adding a new replica to a global table, the read capacity of the source table or replica is used as the initial value unless a replica-level override is explicitly specified.

On-demand mode

For global tables configured for on-demand mode, write capacity is automatically synchronized across all replicas. DynamoDB automatically adjusts capacity based on traffic, and there are no replica-specific read or write capacity settings to manage.

Monitoring global tables

Global tables configured for multi-Region eventual consistency (MREC) publish the `ReplicationLatency` metric to CloudWatch. This metric tracks the elapsed time between when an item is written to a replica table, and when that item appears in another replica in the global table. `ReplicationLatency` is expressed in milliseconds and is emitted for every source and destination Region pair in a global table.

Typical `ReplicationLatency` values depends on the distance between your chosen AWS Regions, as well as other variables like workload type and throughput. For example, a source replica in the US West (N. California) (us-west-1) Region has lower `ReplicationLatency` to the US West (Oregon) (us-west-2) Region compared to the Africa (Cape Town) (af-south-1) Region.

An increasing value for `ReplicationLatency` could indicate that updates from one replica are not propagating to other replica tables in a timely manner. In this case, you can temporarily redirect your application's read and write activity to a different AWS Region.

Global tables configured for multi-Region strong consistency (MRSC) do not publish a `ReplicationLatency` metric.

Considerations for managing global tables

You can't delete a table used to add a new global table replica until 24 hours have elapsed since the new replica was created.

If you disable an AWS Region that contains global table replicas, those replicas are permanently converted to single-Region tables 20 hours after the Region is disabled.

DynamoDB same-account global table

Same-account global tables automatically replicate your DynamoDB table data across AWS Regions within a single AWS account. Same-account global tables provide the simplest model for running multi-Region applications because all replicas share the same account boundary, ownership, and permissions model. When you choose the AWS Regions for your replica tables, global tables handle all replication automatically. Global tables are available in all Regions where DynamoDB is available.

Same-account global tables provide the following benefits:

- Replicate DynamoDB table data automatically across your choice of AWS Regions to locate data closer to your users

- Enable higher application availability during regional isolation or degradation
- Use built-in conflict resolution so you can focus on your application's business logic
- When creating a same-account global table, you can choose either Multi-Region eventual consistency (MREC) or Multi-Region strong consistency (MRSC)

Topics

- How DynamoDB global tables work
- Tutorials: Creating global tables
- DynamoDB global tables security

How DynamoDB global tables work

The following sections describe the concepts and behaviors of global tables in Amazon DynamoDB.

Concepts

Global tables is a DynamoDB feature that replicates table data across AWS Regions.

A *replica table* (or replica) is a DynamoDB table that functions as part of a global table. A global table consists of two or more replica tables across different AWS Regions. Each global table can have only one replica per AWS Region. All replicas in a global table share the same table name, primary key schema, and item data.

When an application writes data to a replica in one Region, DynamoDB automatically replicates the write to all other replicas in the global table. For more information about how to get started with global tables, see Tutorials: Creating global tables.

Versions

There are two versions of DynamoDB global tables available: Version 2019.11.21 (Current) and Version 2017.11.29 (Legacy). You should use Version 2019.11.21 (Current) whenever possible. The information in this documentation section is for Version 2019.11.21 (Current). For more information, see Determining the version of a global table.

Availability

Global tables help improve your business continuity by making it easier to implement a multi-Region high availability architecture. If a workload in a single AWS Region becomes impaired, you

can shift application traffic to a different Region and perform reads and writes to a different replica table in the same global table.

Each replica table in a global table provides the same durability and availability as a single-Region DynamoDB table. Global tables offer a 99.999% availability Service Level Agreement (SLA), compared to 99.99% for single-Region tables.

Consistency modes

When you create a global table, you can configure its consistency mode. Global tables support two consistency modes: multi-Region eventual consistency (MREC), and multi-Region strong consistency (MRSC).

If you do not specify a consistency mode when creating a global table, the global table defaults to multi-Region eventual consistency (MREC). A global table cannot contain replicas configured with different consistency modes. You cannot change a global table's consistency mode after creation.

Multi-Region eventual consistency (MREC)

Multi-Region eventual consistency (MREC) is the default consistency mode for global tables. Item changes in an MREC global table replica are asynchronously replicated to all other replicas, typically within a second or less. In the unlikely event a replica in a MREC global table becomes isolated or impaired, any data not yet replicated to other Regions will be replicated when the replica becomes healthy.

If the same item is modified in multiple Regions simultaneously, DynamoDB will resolve the conflict by using the modification with the latest internal timestamp on a per-item basis, referred to as a "last writer wins" conflict resolution method. An item will eventually converge in all replicas to the version created by the last write.

Strongly consistent read operations return the latest version of an item if that item was last updated in the Region where the read occurred, but may return stale data if the item was last updated in a different Region. Conditional writes evaluate the condition expression against the version of the item in the Region.

You create a MREC global table by adding a replica to an existing DynamoDB table. Adding a replica has no performance impact on existing single-Region DynamoDB tables or global table replicas. You can add replicas to a MREC global table to expand the number of Regions where data is replicated, or remove replicas from an MREC global table if they are no longer needed. An MREC global table can have a replica in any Region where DynamoDB is available, and can have as many replicas as there are Regions in the AWS partition.

Multi-Region strong consistency (MRSC)

You can configure multi-Region strong consistency (MRSC) mode when you create a global table. Item changes in an MRSC global table replica are synchronously replicated to at least one other Region before the write operation returns a successful response. Strongly consistent read operations on any MRSC replica always return the latest version of an item. Conditional writes always evaluate the condition expression against the latest version of an item.

A MRSC global table must be deployed in exactly three Regions. You can configure a MRSC global table with three replicas, or with two replicas and one witness. A witness is a component of a MRSC global table that contains data written to global table replicas and provides an optional alternative to a full replica while supporting MRSC's availability architecture. You cannot perform read or write operations on a witness. A witness is located in a different Region than the two replicas. When creating a MRSC global table, you choose the Regions for both your replicas and the witness deployment at MRSC table creation time. You can determine whether and in which Region a MRSC global table has a witness configured from the output of the `DescribeTable` API. The witness is owned and managed by DynamoDB, and the witness will not appear in your AWS account in the Region where it is configured.

MRSC global tables are available in the following Region sets: US Region set (US East N. Virginia, US East Ohio, US West Oregon), EU Region set (Europe Ireland, Europe London, Europe Paris, Europe Frankfurt), and AP Region set (Asia Pacific Tokyo, Asia Pacific Seoul, and Asia Pacific Osaka). MRSC global tables cannot span Region sets (e.g. a MRSC global table cannot contain replicas from both US and EU Region sets).

You create a MRSC global table by adding one replica and a witness or two replicas to an existing DynamoDB table that contains no data. When converting an existing single-Region table to a MRSC global table, you must ensure that the table is empty. Converting a single-Region table to a MRSC global table with existing items is not supported. Ensure that no data is written into the table during the conversion process. You cannot add additional replicas to an existing MRSC global table. You cannot delete a single replica or a witness from a MRSC global table. You can delete two replicas or delete one replica and a witness from a MRSC global table, converting the remaining replica to a single-Region DynamoDB table.

A write operation fails with a `ReplicatedWriteConflictException` when it attempts to modify an item that is already being modified in another Region. Writes that fail with the `ReplicatedWriteConflictException` can be retried and will succeed if the item is no longer being modified in another Region.

The following considerations apply to MRSC global tables:

- Time to Live (TTL) is not supported for MRSC global tables.
- Local secondary indexes (LSIs) are not supported for MRSC global tables.
- CloudWatch Contributor Insights information is only reported for the Region in which an operation occurred.

Choosing a consistency mode

The key criteria for choosing a multi-Region consistency mode is whether your application prioritizes lower latency writes and strongly consistent reads, or prioritizes global strong consistency.

MREC global tables will have lower write and strongly consistent read latencies compared to MRSC global tables. MREC global tables have a Recovery Point Objective (RPO) equal to the replication delay between replicas, usually a few seconds depending on the replica Regions.

You should use the MREC mode when:

- Your application can tolerate stale data returned from strongly consistent read operations if that data was updated in another Region.
- You prioritize lower write and strongly consistent read latencies over multi-Region read consistency.
- Your multi-Region high availability strategy can tolerate an RPO greater than zero.

MRSC global tables will have higher write and strongly consistent read latencies compared to MREC global tables. MRSC global tables support a Recovery Point Objective (RPO) of zero.

You should use the MRSC mode when:

- You need strongly consistent reads across multiple Regions.
- You prioritize global read consistency over lower write latency.
- Your multi-Region high availability strategy requires an RPO of zero.

Monitoring global tables

Global tables configured for multi-Region eventual consistency (MREC) publish the `ReplicationLatency` metric to CloudWatch. This metric tracks the elapsed time between when

an item is written to a replica table, and when that item appears in another replica in the global table. `ReplicationLatency` is expressed in milliseconds and is emitted for every source and destination Region pair in a global table.

Typical `ReplicationLatency` values depends on the distance between your chosen AWS Regions, as well as other variables like workload type and throughput. For example, a source replica in the US West (N. California) (us-west-1) Region has lower `ReplicationLatency` to the US West (Oregon) (us-west-2) Region compared to the Africa (Cape Town) (af-south-1) Region.

An increasing value for `ReplicationLatency` could indicate that updates from one replica are not propagating to other replica tables in a timely manner. In this case, you can temporarily redirect your application's read and write activity to a different AWS Region.

Global tables configured for multi-Region strong consistency (MRSC) do not publish a `ReplicationLatency` metric.

Fault injection testing

Both MREC and MRSC global tables integrate with AWS Fault Injection Service (AWS FIS), a fully managed service for running controlled fault injection experiments to improve an application's resilience. Using AWS FIS, you can:

- Create experiment templates that define specific failure scenarios.
- Inject failures to validate application resilience by simulating Region isolation (that is, pausing replication to and from a selected replica) to test error handling, recovery mechanisms, and multi-Region traffic shift behavior when one AWS Region experiences disruption.

For example, in a global table with replicas in US East (N. Virginia), US East (Ohio), and US West (Oregon), you can run an experiment in US East (Ohio) to test region isolation there while US East (N. Virginia) and US West (Oregon) continue normal operations. This controlled testing helps you identify and resolve potential issues before they affect production workloads.

See Action targets in the *AWS FIS user guide* for a complete list of AWS FIS supported actions and Cross-Region Connectivity to pause DynamoDB replication between regions.

For information about Amazon DynamoDB global table actions available in AWS FIS, see DynamoDB global tables actions reference in the *AWS FIS User Guide*.

To get started running fault injection experiments, see Planning your AWS FIS experiments in the AWS FIS user guide.

> **Note**
>
> During AWS FIS experiments in MRSC, eventually consistent reads are permitted, but table setting updates - such as changing billing mode or configuring table throughput - are not allowed, similar to MREC. Please check the CloudWatch metric `FaultInjectionServiceInducedErrors` for additional details regarding the error code.

Time To Live (TTL)

Global tables configured for MREC support configuring Time To Live (TTL) deletion. TTL settings are automatically synchronized for all replicas in a global table. When TTL deletes an item from a replica in a Region, the delete is replicated to all other replicas in the global table. TTL does not consume write capacity, so you are not charged for the TTL delete in the Region where the delete occurred. However, you are charged for the replicated delete in each other region with a replica in the global table.

TTL delete replication consumes write capacity on the replicas to which the delete is being replicated. Replicas configured for provisioned capacity may throttle requests if the combination of write throughput and TTL delete throughput is higher than the provisioned write capacity.

Global tables configured for multi-Region strong consistency (MRSC) do not support configuring Time To Live (TTL) deletion.

Streams

Global tables configured for multi-Region eventual consistency (MREC) replicate changes by reading those changes from a DynamoDB Stream on a replica table and applying that change to all other replica tables. Streams are therefore enabled by default on all replicas in an MREC global table, and cannot be disabled on those replicas. The MREC replication process may combine multiple changes in a short period of time into a single replicated write, resulting in each replica's Stream containing slightly different records. Streams records on MREC replicas are always ordered on a per-item basis, but ordering between items may differ between replicas.

Global tables configured for multi-Region strong consistency (MRSC) do not use DynamoDB Streams for replication, so Streams are not enabled by default on MRSC replicas. You can enable Streams on an MRSC replica. Streams records on MRSC replicas are identical for every replica, including Stream record ordering.

If you want to write an application that processes Streams records for changes that occurred in a particular Region but not other Regions in a global table, you can add an attribute to each item that defines in which Region the change for that item occurred. You can use this attribute to filter Streams records for changes that occurred in other Regions, including the use of Lambda event filters to only invoke Lambda functions for changes in a specific Region.

Transactions

On a global table configured for MREC, DynamoDB transaction operations (`TransactWriteItems` and `TransactGetItems`) are only atomic within the Region where the operation was invoked. Transactional writes are not replicated as a unit across Regions, meaning only some of the writes in a transaction may be returned by read operations in other replicas at a given point in time.

For example, if you have a global table with replicas in the US East (Ohio) and US West (Oregon) Regions and perform a `TransactWriteItems` operation in the US East (Ohio) Region, you may observe partially completed transactions in the US West (Oregon) Region as changes are replicated. Changes will only be replicated to other Regions once they've been committed in the source Region.

Global tables configured for multi-Region strong consistency (MRSC) do not support transaction operations, and will return an error if those operations are invoked on an MRSC replica.

Read and write throughput

Provisioned mode

Replication consumes write capacity. Replicas configured for provisioned capacity may throttle requests if the combination of application write throughput and replication write throughput exceeds the provisioned write capacity. For global tables using provisioned mode, auto scaling settings for both read and write capacities are synchronized between replicas.

You can independently configure read capacity settings for each replica in a global table by using the `ProvisionedThroughputOverride` parameter at the replica level. By default, changes to provisioned read capacity are applied to all replicas in the global table. When adding a new replica to a global table, the read capacity of the source table or replica is used as the initial value unless a replica-level override is explicitly specified.

On-demand mode

For global tables configured for on-demand mode, write capacity is automatically synchronized across all replicas. DynamoDB automatically adjusts capacity based on traffic, and there are no replica-specific read or write capacity settings to manage.

Settings synchronization

Settings in DynamoDB global tables are configuration parameters that control various aspects of table behavior and replication. These settings are managed through the DynamoDB control plane APIs and can be configured when creating or modifying global tables. Global tables automatically synchronize certain settings across all replicas to maintain consistency, while allowing flexibility for region-specific optimizations. Understanding which settings synchronize and how they behave helps you configure your global table effectively. The settings fall into three main categories based on how they are synchronized across replicas.

The following settings are always synchronized between replicas in a global table:

- Capacity mode (provisioned capacity or on-demand)
- Table provisioned write capacity
- Table write auto scaling
- Attribute definition of key schema
- Global Secondary Index (GSI) definition
- GSI provisioned write capacity
- GSI write auto scaling
- Server-side Encryption (SSE) type
- Streams definition in MREC mode
- Time To Live (TTL)
- Warm Throughput
- On-demand maximum write throughput

The following settings are synchronized between replicas, but can be overridden on a per-replica basis:

- Table provisioned read capacity
- Table read auto scaling

- GSI provisioned read capacity
- GSI read auto scaling
- Table Class
- On-demand maximum read throughput

> **Note**
>
> Overridable setting values are changed if the setting is modified on any other replica. As an example, you have a MREC global table with replicas in US East (N. Virginia) and US West (Oregon). The US East (N. Virginia) replica has provisioned read throughput set to 200 RCUs. The replica in US West (Oregon) has a provisioned read throughput override set to 100 RCUs. If you update the provisioned read throughput setting on the US East (N. Virginia) replica from 200 RCUs to 300 RCUs, the new provisioned read throughput value will also be applied to the replica in US West (Oregon). This changes the provisioned read throughput setting for the US West (Oregon) replica from the overridden value of 100 RCUs to the new value of 300 RCUs.

The following settings are never synchronized between replicas:

- Deletion protection
- Point-in-time Recovery
- Tags
- Table CloudWatch Contributor Insights enablement
- GSI CloudWatch Contributor Insights enablement
- Kinesis Data Streams definition
- Resource Policies
- Streams definition in MRSC mode

All other settings are not synchronized between replicas.

DynamoDB Accelerator (DAX)

Writes to global table replicas bypass DynamoDB Accelerator (DAX), updating DynamoDB directly. As a result, DAX caches can become stale as writes are not updating the DAX cache. DAX caches configured for global table replicas will only be refreshed when the cache TTL expires.

Considerations for managing global tables

You can't delete a table used to add a new global table replica until 24 hours have elapsed since the new replica was created.

If you disable an AWS Region that contains global table replicas, those replicas are permanently converted to single-Region tables 20 hours after the Region is disabled.

Tutorials: Creating global tables

This section provides step-by-step instructions for creating DynamoDB global tables configured for your preferred consistency mode. Choose either Multi-Region Eventual Consistency (MREC) or Multi-Region Strong Consistency (MRSC) modes based on your application's requirements.

MREC global tables provide lower write latency with eventual consistency across AWS Regions. MRSC global tables provide strongly consistent reads across Regions with slightly higher write latencies than MREC. Choose the consistency mode that best meets your application's needs for data consistency, latency, and availability.

Topics

- Creating a global table configured for MREC
- Creating a global table configured for MRSC

Creating a global table configured for MREC

This section shows how to create a global table with Multi-Region Eventual Consistency (MREC) mode. MREC is the default consistency mode for global tables and provides low-latency writes with asynchronous replication across AWS Regions. Changes made to an item in one region are typically replicated to all other regions within a second. This makes MREC ideal for applications that prioritize low write latency and can tolerate brief periods where different Regions may return slightly different versions of data.

You can create MREC global tables with replicas in any AWS Region where DynamoDB is available and add or remove replicas at any time. The following examples show how to create an MREC global table with replicas in multiple regions.

Creating a MREC global table using the DynamoDB Console

Follow these steps to create a global table using the AWS Management Console. The following example creates a global table with replica tables in the United States and Europe.

1. Sign in to the AWS Management Console and open the DynamoDB console at https://console.aws.amazon.com/dynamodb/.
2. For this example, choose **US East (Ohio)** from the Region selector in the navigation bar.
3. In the navigation pane on the left side of the console, choose **Tables**.
4. Choose **Create Table**.
5. On the **Create table** page:
    a. For **Table name**, enter **`Music`**.
    b. For **Partition key**, enter **`Artist`**.
    c. For **Sort key**, enter **`SongTitle`**.
    d. Keep the other default settings and choose **Create table**.

    This new table serves as the first replica table in a new global table. It is the prototype for other replica tables that you add later.
6. After the table becomes active:
    a. Select the **Music** table from the tables list.
    b. Choose the **Global tables** tab.
    c. Choose **Create replica**.
7. From the **Available replication Regions** dropdown list, choose **US West (Oregon) us-west-2**.

    The console ensures that a table with the same name doesn't exist in the selected Region. If a table with the same name does exist, you must delete the existing table before you can create a new replica table in that Region.
8. Choose **Create replica**. This starts the table creation process in the US West (Oregon) us-west-2 Region.

The **Global tables** tab for the **Music** table (and for any other replica tables) shows that the table has been replicated in multiple Regions.

9. Add another region by repeating the previous steps, but choose **Europe (Frankfurt) eu-central-1** as the region.
10. To test replication:
    a. Make sure you're using the AWS Management Console in the US East (Ohio) Region.
    b. Choose **Explore table items**.
    c. Choose **Create item**.
    d. Enter **`item_1`** for **Artist** and **`Song Value 1`** for **SongTitle**.
    e. Choose **Create item**.
11. Verify replication by switching to the other regions:
    a. From the Region selector in the upper-right corner, choose **Europe (Frankfurt)**.
    b. Verify that the **Music** table contains the item you created.
    c. Repeat the verification for **US West (Oregon)**.

Creating a MREC global table using the AWS CLI or Java

CLI

The following code example shows how to manage DynamoDB global tables with multi-Region replication with eventual consistency (MREC).

- Create a table with multi-Region replication (MREC).
- Put and get items from replica tables.
- Remove replicas one-by-one.
- Clean up by deleting the table.

AWS CLI with Bash script

Create a table with multi-Region replication.

```
# Step 1: Create a new table (MusicTable) in US East (Ohio), with DynamoDB
 Streams enabled (NEW_AND_OLD_IMAGES)
aws dynamodb create-table \
```

```
    --table-name MusicTable \
    --attribute-definitions \
        AttributeName=Artist,AttributeType=S \
        AttributeName=SongTitle,AttributeType=S \
    --key-schema \
        AttributeName=Artist,KeyType=HASH \
        AttributeName=SongTitle,KeyType=RANGE \
    --billing-mode PAY_PER_REQUEST \
    --stream-specification StreamEnabled=true,StreamViewType=NEW_AND_OLD_IMAGES \
    --region us-east-2

# Step 2: Create an identical MusicTable table in US East (N. Virginia)
aws dynamodb update-table --table-name MusicTable --cli-input-json \
'{
  "ReplicaUpdates":
  [
    {
      "Create": {
        "RegionName": "us-east-1"
      }
    }
  ]
}' \
--region us-east-2

# Step 3: Create a table in Europe (Ireland)
aws dynamodb update-table --table-name MusicTable --cli-input-json \
'{
  "ReplicaUpdates":
  [
    {
      "Create": {
        "RegionName": "eu-west-1"
      }
    }
  ]
}' \
--region us-east-2
```

Describe the multi-Region table.

```
# Step 4: View the list of replicas created using describe-table
```

```
aws dynamodb describe-table \
    --table-name MusicTable \
    --region us-east-2 \
    --query 'Table.
{TableName:TableName,TableStatus:TableStatus,MultiRegionConsistency:MultiRegionConsistenc
{Region:RegionName,Status:ReplicaStatus}}'
```

Put items in a replica table.

```
# Step 5: To verify that replication is working, add a new item to the Music
 table in US East (Ohio)
aws dynamodb put-item \
    --table-name MusicTable \
    --item '{"Artist": {"S":"item_1"},"SongTitle": {"S":"Song Value 1"}}' \
    --region us-east-2
```

Get items from replica tables.

```
# Step 6: Wait for a few seconds, and then check to see whether the item has
 been
# successfully replicated to US East (N. Virginia) and Europe (Ireland)
aws dynamodb get-item \
    --table-name MusicTable \
    --key '{"Artist": {"S":"item_1"},"SongTitle": {"S":"Song Value 1"}}' \
    --region us-east-1

aws dynamodb get-item \
    --table-name MusicTable \
    --key '{"Artist": {"S":"item_1"},"SongTitle": {"S":"Song Value 1"}}' \
    --region eu-west-1
```

Remove replicas.

```
# Step 7: Delete the replica table in Europe (Ireland) Region
aws dynamodb update-table --table-name MusicTable --cli-input-json \
'{
  "ReplicaUpdates":
  [
    {
```

```
      "Delete": {
        "RegionName": "eu-west-1"
      }
    }
  ]
}' \
--region us-east-2

# Delete the replica table in US East (N. Virginia) Region
aws dynamodb update-table --table-name MusicTable --cli-input-json \
'{
  "ReplicaUpdates":
  [
    {
      "Delete": {
        "RegionName": "us-east-1"
      }
    }
  ]
}' \
--region us-east-2
```

Clean up by deleting the table.

```
# Clean up: Delete the primary table
aws dynamodb delete-table --table-name MusicTable --region us-east-2

echo "Global table demonstration complete."
```

- For API details, see the following topics in *AWS CLI Command Reference*.
  - CreateTable
  - DeleteTable
  - DescribeTable
  - GetItem
  - PutItem
  - UpdateTable

Java

The following code example shows how to create and manage DynamoDB global tables with replicas across multiple Regions.

- Create a table with Global Secondary Index and DynamoDB Streams.
- Add replicas in different Regions to create a global table.
- Remove replicas from a global table.
- Add test items to verify replication across Regions.
- Describe global table configuration and replica status.

SDK for Java 2.x

Create a table with Global Secondary Index and DynamoDB Streams using AWS SDK for Java 2.x.

```
   public static CreateTableResponse createTableWithGSI(
       final DynamoDbClient dynamoDbClient, final String tableName, final String
indexName) {

       if (dynamoDbClient == null) {
           throw new IllegalArgumentException("DynamoDB client cannot be null");
       }
       if (tableName == null || tableName.trim().isEmpty()) {
           throw new IllegalArgumentException("Table name cannot be null or
empty");
       }
       if (indexName == null || indexName.trim().isEmpty()) {
           throw new IllegalArgumentException("Index name cannot be null or
empty");
       }

       try {
           LOGGER.info("Creating table: " + tableName + " with GSI: " +
indexName);

           CreateTableRequest createTableRequest = CreateTableRequest.builder()
               .tableName(tableName)
               .attributeDefinitions(
                   AttributeDefinition.builder()
                       .attributeName("Artist")
```

```
                        .attributeType(ScalarAttributeType.S)
                        .build(),
                    AttributeDefinition.builder()
                        .attributeName("SongTitle")
                        .attributeType(ScalarAttributeType.S)
                        .build())
                .keySchema(
                    KeySchemaElement.builder()
                        .attributeName("Artist")
                        .keyType(KeyType.HASH)
                        .build(),
                    KeySchemaElement.builder()
                        .attributeName("SongTitle")
                        .keyType(KeyType.RANGE)
                        .build())
                .billingMode(BillingMode.PAY_PER_REQUEST)
                .globalSecondaryIndexes(GlobalSecondaryIndex.builder()
                    .indexName(indexName)
                    .keySchema(KeySchemaElement.builder()
                        .attributeName("SongTitle")
                        .keyType(KeyType.HASH)
                        .build())
                    .projection(

Projection.builder().projectionType(ProjectionType.ALL).build())
                    .build())
                .streamSpecification(StreamSpecification.builder()
                    .streamEnabled(true)
                    .streamViewType(StreamViewType.NEW_AND_OLD_IMAGES)
                    .build())
                .build();

            CreateTableResponse response =
dynamoDbClient.createTable(createTableRequest);
            LOGGER.info("Table creation initiated. Status: "
                + response.tableDescription().tableStatus());

            return response;

        } catch (DynamoDbException e) {
            LOGGER.severe("Failed to create table: " + tableName + " - " +
e.getMessage());
            throw e;
        }
```

```
    }
```

Wait for a table to become active using AWS SDK for Java 2.x.

```
    public static void waitForTableActive(final DynamoDbClient dynamoDbClient,
final String tableName) {

        if (dynamoDbClient == null) {
            throw new IllegalArgumentException("DynamoDB client cannot be null");
        }
        if (tableName == null || tableName.trim().isEmpty()) {
            throw new IllegalArgumentException("Table name cannot be null or
empty");
        }

        try {
            LOGGER.info("Waiting for table to become active: " + tableName);

            try (DynamoDbWaiter waiter =
                DynamoDbWaiter.builder().client(dynamoDbClient).build()) {
                DescribeTableRequest request =
                    DescribeTableRequest.builder().tableName(tableName).build();

                waiter.waitUntilTableExists(request);
                LOGGER.info("Table is now active: " + tableName);
            }

        } catch (DynamoDbException e) {
            LOGGER.severe("Failed to wait for table to become active: " +
tableName + " - " + e.getMessage());
            throw e;
        }
    }
```

Add a replica to create or extend a global table using AWS SDK for Java 2.x.

```
    public static UpdateTableResponse addReplica(
        final DynamoDbClient dynamoDbClient,
        final String tableName,
        final Region replicaRegion,
        final String indexName,
```

```
        final Long readCapacity) {

        if (dynamoDbClient == null) {
            throw new IllegalArgumentException("DynamoDB client cannot be null");
        }
        if (tableName == null || tableName.trim().isEmpty()) {
            throw new IllegalArgumentException("Table name cannot be null or
empty");
        }
        if (replicaRegion == null) {
            throw new IllegalArgumentException("Replica region cannot be null");
        }
        if (indexName == null || indexName.trim().isEmpty()) {
            throw new IllegalArgumentException("Index name cannot be null or
empty");
        }
        if (readCapacity == null || readCapacity <= 0) {
            throw new IllegalArgumentException("Read capacity must be a positive
number");
        }

        try {
            LOGGER.info("Adding replica in region: " + replicaRegion.id() + " for
table: " + tableName);

            // Create a ReplicationGroupUpdate for adding a replica
            ReplicationGroupUpdate replicationGroupUpdate =
ReplicationGroupUpdate.builder()
                .create(builder -> builder.regionName(replicaRegion.id())
                    .globalSecondaryIndexes(ReplicaGlobalSecondaryIndex.builder()
                        .indexName(indexName)

.provisionedThroughputOverride(ProvisionedThroughputOverride.builder()
                            .readCapacityUnits(readCapacity)
                            .build())
                        .build())
                    .build())
                .build();

            UpdateTableRequest updateTableRequest = UpdateTableRequest.builder()
                .tableName(tableName)
                .replicaUpdates(replicationGroupUpdate)
                .build();
```

```
            UpdateTableResponse response =
dynamoDbClient.updateTable(updateTableRequest);
            LOGGER.info("Replica addition initiated in region: " +
replicaRegion.id());

            return response;

        } catch (DynamoDbException e) {
            LOGGER.severe("Failed to add replica in region: " +
replicaRegion.id() + " - " + e.getMessage());
            throw e;
        }
    }
```

Remove a replica from a global table using AWS SDK for Java 2.x.

```
    public static UpdateTableResponse removeReplica(
        final DynamoDbClient dynamoDbClient, final String tableName, final Region
replicaRegion) {

        if (dynamoDbClient == null) {
            throw new IllegalArgumentException("DynamoDB client cannot be null");
        }
        if (tableName == null || tableName.trim().isEmpty()) {
            throw new IllegalArgumentException("Table name cannot be null or
empty");
        }
        if (replicaRegion == null) {
            throw new IllegalArgumentException("Replica region cannot be null");
        }

        try {
            LOGGER.info("Removing replica in region: " + replicaRegion.id() + "
for table: " + tableName);

            // Create a ReplicationGroupUpdate for removing a replica
            ReplicationGroupUpdate replicationGroupUpdate =
ReplicationGroupUpdate.builder()
                .delete(builder ->
builder.regionName(replicaRegion.id()).build())
                .build();
```

```
            UpdateTableRequest updateTableRequest = UpdateTableRequest.builder()
                .tableName(tableName)
                .replicaUpdates(replicationGroupUpdate)
                .build();

            UpdateTableResponse response =
dynamoDbClient.updateTable(updateTableRequest);
            LOGGER.info("Replica removal initiated in region: " +
replicaRegion.id());

            return response;

        } catch (DynamoDbException e) {
            LOGGER.severe("Failed to remove replica in region: " +
replicaRegion.id() + " - " + e.getMessage());
            throw e;
        }
    }
```

Add test items to verify replication using AWS SDK for Java 2.x.

```
    public static PutItemResponse putTestItem(
        final DynamoDbClient dynamoDbClient, final String tableName, final String
artist, final String songTitle) {

        if (dynamoDbClient == null) {
            throw new IllegalArgumentException("DynamoDB client cannot be null");
        }
        if (tableName == null || tableName.trim().isEmpty()) {
            throw new IllegalArgumentException("Table name cannot be null or
empty");
        }
        if (artist == null || artist.trim().isEmpty()) {
            throw new IllegalArgumentException("Artist cannot be null or empty");
        }
        if (songTitle == null || songTitle.trim().isEmpty()) {
            throw new IllegalArgumentException("Song title cannot be null or
empty");
        }

        try {
            LOGGER.info("Adding test item to table: " + tableName);
```

```
           Map<String,
software.amazon.awssdk.services.dynamodb.model.AttributeValue> item = new
HashMap<>();
           item.put(
               "Artist",

software.amazon.awssdk.services.dynamodb.model.AttributeValue.builder()
                   .s(artist)
                   .build());
           item.put(
               "SongTitle",

software.amazon.awssdk.services.dynamodb.model.AttributeValue.builder()
                   .s(songTitle)
                   .build());

           PutItemRequest putItemRequest =
               PutItemRequest.builder().tableName(tableName).item(item).build();

           PutItemResponse response = dynamoDbClient.putItem(putItemRequest);
           LOGGER.info("Test item added successfully");

           return response;

       } catch (DynamoDbException e) {
           LOGGER.severe("Failed to add test item to table: " + tableName + " -
" + e.getMessage());
           throw e;
       }
   }
```

Describe global table configuration and replicas using AWS SDK for Java 2.x.

```
   public static DescribeTableResponse describeTable(final DynamoDbClient
dynamoDbClient, final String tableName) {

       if (dynamoDbClient == null) {
           throw new IllegalArgumentException("DynamoDB client cannot be null");
       }
       if (tableName == null || tableName.trim().isEmpty()) {
```

```
            throw new IllegalArgumentException("Table name cannot be null or
empty");
        }

        try {
            LOGGER.info("Describing table: " + tableName);

            DescribeTableRequest request =
                DescribeTableRequest.builder().tableName(tableName).build();

            DescribeTableResponse response =
dynamoDbClient.describeTable(request);

            LOGGER.info("Table status: " + response.table().tableStatus());
            if (response.table().replicas() != null
                && !response.table().replicas().isEmpty()) {
                LOGGER.info("Number of replicas: " +
response.table().replicas().size());
                response.table()
                    .replicas()
                    .forEach(replica -> LOGGER.info(
                        "Replica region: " + replica.regionName() + ", Status: "
+ replica.replicaStatus()));
            }

            return response;

        } catch (ResourceNotFoundException e) {
            LOGGER.severe("Table not found: " + tableName + " - " +
e.getMessage());
            throw e;
        } catch (DynamoDbException e) {
            LOGGER.severe("Failed to describe table: " + tableName + " - " +
e.getMessage());
            throw e;
        }
    }
```

Complete example of global table operations using AWS SDK for Java 2.x.

```
    public static void exampleUsage(final Region sourceRegion, final Region
replicaRegion) {
```

```
        String tableName = "Music";
        String indexName = "SongTitleIndex";
        Long readCapacity = 15L;

        // Create DynamoDB client for the source region
        try (DynamoDbClient dynamoDbClient =
            DynamoDbClient.builder().region(sourceRegion).build()) {

            try {
                // Step 1: Create the initial table with GSI and streams
                LOGGER.info("Step 1: Creating table in source region: " +
sourceRegion.id());
                createTableWithGSI(dynamoDbClient, tableName, indexName);

                // Step 2: Wait for table to become active
                LOGGER.info("Step 2: Waiting for table to become active");
                waitForTableActive(dynamoDbClient, tableName);

                // Step 3: Add replica in destination region
                LOGGER.info("Step 3: Adding replica in region: " +
replicaRegion.id());
                addReplica(dynamoDbClient, tableName, replicaRegion, indexName,
readCapacity);

                // Step 4: Wait a moment for replica creation to start
                Thread.sleep(5000);

                // Step 5: Describe table to view replica information
                LOGGER.info("Step 5: Describing table to view replicas");
                describeTable(dynamoDbClient, tableName);

                // Step 6: Add a test item to verify replication
                LOGGER.info("Step 6: Adding test item to verify replication");
                putTestItem(dynamoDbClient, tableName, "TestArtist", "TestSong");

                LOGGER.info("Global table setup completed successfully!");
                LOGGER.info("You can verify replication by checking the item in
region: " + replicaRegion.id());

                // Step 7: Remove replica and clean up table
                LOGGER.info("Step 7: Removing replica from region: " +
replicaRegion.id());
                removeReplica(dynamoDbClient, tableName, replicaRegion);
```

```
                DeleteTableResponse deleteTableResponse =
dynamoDbClient.deleteTable(
                    DeleteTableRequest.builder().tableName(tableName).build());
                LOGGER.info("MREC global table demonstration completed
successfully!");

            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
                throw new RuntimeException("Thread was interrupted", e);
            } catch (DynamoDbException e) {
                LOGGER.severe("DynamoDB operation failed: " + e.getMessage());
                throw e;
            }
        }
    }
```

- For API details, see the following topics in *AWS SDK for Java 2.x API Reference*.
  - CreateTable
  - DescribeTable
  - PutItem
  - UpdateTable

Creating a global table configured for MRSC

This section shows you how to create a Multi-Region Strong Consistency (MRSC) global table. MRSC global tables synchronously replicate item changes across Regions, ensuring that strongly consistent read operations on any replica always return the latest version of an item. When converting a single-Region table to a MRSC global table, you must ensure that the table is empty. Converting a single-Region table to a MRSC global table with existing items is not supported. Ensure that no data is written into the table during the conversion process.

You can configure a MRSC global table with three replicas, or two replicas and one witness. When creating a MRSC global table, you choose the Regions where replicas and an optional witness are deployed. The following example creates an MRSC global table with replicas in the US East (N. Virginia) and US East (Ohio) Regions, with a witness in the US West (Oregon) Region.

> **Note**
>
> Before creating a global table, verify that the Service Quota throughput limits are consistent across all target Regions, as this is required to create a global table. For more information about global table throughput limits, see Global tables quotas.

Creating a MRSC global table using the DynamoDB Console

Follow these steps to create about MRSC global table using the AWS Management Console.

1. Sign in to the AWS Management Console and open the DynamoDB console at https://console.aws.amazon.com/dynamodb/.
2. From the Region selector in the navigation bar, choose a Region where global tables with MRSC is supported, such as **`us-east-2`**.
3. In the navigation pane, choose **Tables**.
4. Choose **Create table**.
5. On the **Create table** page:
    a. For **Table name**, enter **`Music`**.
    b. For **Partition key**, enter **`Artist`** and keep the default **String** type.
    c. For **Sort key**, enter **`SongTitle`** and keep the default **String** type.
    d. Keep the other default settings and choose **Create table**

    This new table serves as the first replica table in a new global table. It is the prototype for other replica tables that you add later.
6. Wait for the table to become active, then select it from the tables list.
7. Choose the **Global tables** tab, then choose **Create replica**.
8. On the **Create replica** page:
    a. Under **Multi-Region Consistency**, choose **Strong consistency**.
    b. For **Replication Region 1**, choose **`US East (N. Virginia) us-east-1`**.
    c. For **Replication Region 2**, choose **`US West (Oregon) us-west-2`**.
    d. Check **Configure as Witness** for the US West (Oregon) region.
    e. Choose **Create replicas**.

9. Wait for the replica and witness creation process to complete. The replica status will show as **Active** when the table is ready to use.

Creating a MRSC global table using the AWS CLI or Java

Before you start, ensure that your IAM principal has the required permissions to create a MRSC global table with a witness Region.

The following sample IAM policy allows you to create a DynamoDB table (`MusicTable`) in US East (Ohio) with a replica in US East (N. Virginia) and a witness Region in US West (Oregon):

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "dynamodb:CreateTable",
                "dynamodb:CreateTableReplica",
                "dynamodb:CreateGlobalTableWitness",
                "dynamodb:DescribeTable",
                "dynamodb:UpdateTable",
                "dynamodb:DeleteTable",
                "dynamodb:DeleteTableReplica",
                "dynamodb:DeleteGlobalTableWitness",
                "dynamodb:Scan",
                "dynamodb:Query",
                "dynamodb:UpdateItem",
                "dynamodb:PutItem",
                "dynamodb:GetItem",
                "dynamodb:DeleteItem",
                "dynamodb:BatchWriteItem"
            ],
            "Resource": [
                "arn:aws:dynamodb:us-east-1:123456789012:table/MusicTable",
                "arn:aws:dynamodb:us-east-2:123456789012:table/MusicTable",
                "arn:aws:dynamodb:us-west-2:123456789012:table/MusicTable"
            ]
        },
        {
```

```
            "Effect": "Allow",
            "Action": "iam:CreateServiceLinkedRole",
            "Resource": "arn:aws:iam::*:role/aws-service-role/
replication.dynamodb.amazonaws.com/AWSServiceRoleForDynamoDBReplication",
            "Condition": {
                "StringLike": {
                    "iam:AWSServiceName": "replication.dynamodb.amazonaws.com"
                }
            }
        }
    ]
}
```

The following code examples show how to create and manage DynamoDB global tables with Multi-Region Strong Consistency (MRSC).

- Create a table with Multi-Region Strong Consistency.
- Verify MRSC configuration and replica status.
- Test strong consistency across Regions with immediate reads.
- Perform conditional writes with MRSC guarantees.
- Clean up MRSC global table resources.

Bash

AWS CLI with Bash script

Create a table with Multi-Region Strong Consistency.

```
# Step 1: Create a new table in us-east-2 (primary region for MRSC)
# Note: Table must be empty when enabling MRSC
aws dynamodb create-table \
    --table-name MusicTable \
    --attribute-definitions \
        AttributeName=Artist,AttributeType=S \
        AttributeName=SongTitle,AttributeType=S \
    --key-schema \
        AttributeName=Artist,KeyType=HASH \
        AttributeName=SongTitle,KeyType=RANGE \
    --billing-mode PAY_PER_REQUEST \
    --region us-east-2
```

```
# Wait for table to become active
aws dynamodb wait table-exists --table-name MusicTable --region us-east-2

# Step 2: Add replica and witness with Multi-Region Strong Consistency
# MRSC requires exactly three replicas in supported regions
aws dynamodb update-table \
    --table-name MusicTable \
    --replica-updates '[{"Create": {"RegionName": "us-east-1"}}]' \
    --global-table-witness-updates '[{"Create": {"RegionName": "us-west-2"}}]' \
    --multi-region-consistency STRONG \
    --region us-east-2
```

Verify MRSC configuration and replica status.

```
# Verify the global table configuration and MRSC setting
aws dynamodb describe-table \
    --table-name MusicTable \
    --region us-east-2 \
    --query 'Table.
{TableName:TableName,TableStatus:TableStatus,MultiRegionConsistency:MultiRegionConsistenc
{Region:RegionName,Status:ReplicaStatus}}'
```

Test strong consistency with immediate reads across Regions.

```
# Write an item to the primary region
aws dynamodb put-item \
    --table-name MusicTable \
    --item '{"Artist": {"S":"The Beatles"},"SongTitle": {"S":"Hey Jude"},"Album":
 {"S":"The Beatles 1967-1970"},"Year": {"N":"1968"}}' \
    --region us-east-2

# Read the item from replica region to verify strong consistency (cannot read or
 write to witness)
# No wait time needed - MRSC provides immediate consistency
echo "Reading from us-east-1 (immediate consistency):"
aws dynamodb get-item \
    --table-name MusicTable \
    --key '{"Artist": {"S":"The Beatles"},"SongTitle": {"S":"Hey Jude"}}' \
    --consistent-read \
    --region us-east-1
```

Perform conditional writes with MRSC guarantees.

```
# Perform a conditional update from a different region
# This demonstrates that conditions work consistently across all regions
aws dynamodb update-item \
    --table-name MusicTable \
    --key '{"Artist": {"S":"The Beatles"},"SongTitle": {"S":"Hey Jude"}}' \
    --update-expression "SET #rating = :rating" \
    --condition-expression "attribute_exists(Artist)" \
    --expression-attribute-names '{"#rating": "Rating"}' \
    --expression-attribute-values '{":rating": {"N":"5"}}' \
    --region us-east-1
```

Clean up MRSC global table resources.

```
# Remove replica tables (must be done before deleting the primary table)
aws dynamodb update-table \
    --table-name MusicTable \
    --replica-updates '[{"Delete": {"RegionName": "us-east-1"}}]' \
    --global-table-witness-updates '[{"Delete": {"RegionName": "us-west-2"}}]' \
    --region us-east-2

# Wait for replicas to be deleted
echo "Waiting for replicas to be deleted..."
sleep 30

# Delete the primary table
aws dynamodb delete-table \
    --table-name MusicTable \
    --region us-east-2
```

- For API details, see the following topics in *AWS CLI Command Reference*.
  - CreateTable
  - DeleteTable
  - DescribeTable
  - GetItem

- PutItem
- UpdateItem
- UpdateTable

Java

SDK for Java 2.x

Create a regional table ready for MRSC conversion using AWS SDK for Java 2.x.

```
   public static CreateTableResponse createRegionalTable(final DynamoDbClient
dynamoDbClient, final String tableName) {

       if (dynamoDbClient == null) {
           throw new IllegalArgumentException("DynamoDB client cannot be null");
       }
       if (tableName == null || tableName.trim().isEmpty()) {
           throw new IllegalArgumentException("Table name cannot be null or
empty");
       }

       try {
           LOGGER.info("Creating regional table: " + tableName + " (must be
empty for MRSC)");

           CreateTableRequest createTableRequest = CreateTableRequest.builder()
               .tableName(tableName)
               .attributeDefinitions(
                   AttributeDefinition.builder()
                       .attributeName("Artist")
                       .attributeType(ScalarAttributeType.S)
                       .build(),
                   AttributeDefinition.builder()
                       .attributeName("SongTitle")
                       .attributeType(ScalarAttributeType.S)
                       .build())
               .keySchema(
                   KeySchemaElement.builder()
                       .attributeName("Artist")
                       .keyType(KeyType.HASH)
                       .build(),
                   KeySchemaElement.builder()
```

```
                        .attributeName("SongTitle")
                        .keyType(KeyType.RANGE)
                        .build())
                .billingMode(BillingMode.PAY_PER_REQUEST)
                .build();

            CreateTableResponse response =
dynamoDbClient.createTable(createTableRequest);
            LOGGER.info("Regional table creation initiated. Status: "
                + response.tableDescription().tableStatus());

            return response;

        } catch (DynamoDbException e) {
            LOGGER.severe("Failed to create regional table: " + tableName + " - "
+ e.getMessage());
            throw DynamoDbException.builder()
                .message("Failed to create regional table: " + tableName)
                .cause(e)
                .build();
        }
    }
```

Convert a regional table to MRSC with replicas and witness using AWS SDK for Java 2.x.

```
    public static UpdateTableResponse convertToMRSCWithWitness(
        final DynamoDbClient dynamoDbClient,
        final String tableName,
        final Region replicaRegion,
        final Region witnessRegion) {

        if (dynamoDbClient == null) {
            throw new IllegalArgumentException("DynamoDB client cannot be null");
        }
        if (tableName == null || tableName.trim().isEmpty()) {
            throw new IllegalArgumentException("Table name cannot be null or
empty");
        }
        if (replicaRegion == null) {
            throw new IllegalArgumentException("Replica region cannot be null");
        }
        if (witnessRegion == null) {
```

```
            throw new IllegalArgumentException("Witness region cannot be null");
        }

        try {
            LOGGER.info("Converting table to MRSC with replica in " +
replicaRegion.id() + " and witness in "
                + witnessRegion.id());

            // Create replica update using ReplicationGroupUpdate
            ReplicationGroupUpdate replicaUpdate =
ReplicationGroupUpdate.builder()
                .create(CreateReplicationGroupMemberAction.builder()
                    .regionName(replicaRegion.id())
                    .build())
                .build();

            // Create witness update
            GlobalTableWitnessGroupUpdate witnessUpdate =
GlobalTableWitnessGroupUpdate.builder()
                .create(CreateGlobalTableWitnessGroupMemberAction.builder()
                    .regionName(witnessRegion.id())
                    .build())
                .build();

            UpdateTableRequest updateTableRequest = UpdateTableRequest.builder()
                .tableName(tableName)
                .replicaUpdates(List.of(replicaUpdate))
                .globalTableWitnessUpdates(List.of(witnessUpdate))
                .multiRegionConsistency(MultiRegionConsistency.STRONG)
                .build();

            UpdateTableResponse response =
dynamoDbClient.updateTable(updateTableRequest);
            LOGGER.info("MRSC conversion initiated. Status: "
                + response.tableDescription().tableStatus());
            LOGGER.info("UpdateTableResponse full object: " + response);
            return response;

        } catch (DynamoDbException e) {
            LOGGER.severe("Failed to convert table to MRSC: " + tableName + " - "
+ e.getMessage());
            throw DynamoDbException.builder()
                .message("Failed to convert table to MRSC: " + tableName)
                .cause(e)
```

```
                .build();
        }
    }
```

Describe an MRSC global table configuration using AWS SDK for Java 2.x.

```
   public static DescribeTableResponse describeMRSCTable(final DynamoDbClient
dynamoDbClient, final String tableName) {

       if (dynamoDbClient == null) {
           throw new IllegalArgumentException("DynamoDB client cannot be null");
       }
       if (tableName == null || tableName.trim().isEmpty()) {
           throw new IllegalArgumentException("Table name cannot be null or
empty");
       }

       try {
           LOGGER.info("Describing MRSC global table: " + tableName);

           DescribeTableRequest request =
               DescribeTableRequest.builder().tableName(tableName).build();

           DescribeTableResponse response =
dynamoDbClient.describeTable(request);

           LOGGER.info("Table status: " + response.table().tableStatus());
           LOGGER.info("Multi-region consistency: " +
response.table().multiRegionConsistency());

           if (response.table().replicas() != null
               && !response.table().replicas().isEmpty()) {
               LOGGER.info("Number of replicas: " +
response.table().replicas().size());
               response.table()
                   .replicas()
                   .forEach(replica -> LOGGER.info(
                       "Replica region: " + replica.regionName() + ", Status: "
+ replica.replicaStatus()));
           }

           if (response.table().globalTableWitnesses() != null
```

```
                && !response.table().globalTableWitnesses().isEmpty()) {
                LOGGER.info("Number of witnesses: "
                    + response.table().globalTableWitnesses().size());
                response.table()
                    .globalTableWitnesses()
                    .forEach(witness -> LOGGER.info(
                        "Witness region: " + witness.regionName() + ", Status: "
+ witness.witnessStatus()));
            }

            return response;

        } catch (ResourceNotFoundException e) {
            LOGGER.severe("Table not found: " + tableName + " - " +
e.getMessage());
            throw DynamoDbException.builder()
                .message("Table not found: " + tableName)
                .cause(e)
                .build();
        } catch (DynamoDbException e) {
            LOGGER.severe("Failed to describe table: " + tableName + " - " +
e.getMessage());
            throw DynamoDbException.builder()
                .message("Failed to describe table: " + tableName)
                .cause(e)
                .build();
        }
    }
```

Add test items to verify MRSC strong consistency using AWS SDK for Java 2.x.

```
    public static PutItemResponse putTestItem(
        final DynamoDbClient dynamoDbClient,
        final String tableName,
        final String artist,
        final String songTitle,
        final String album,
        final String year) {

        if (dynamoDbClient == null) {
            throw new IllegalArgumentException("DynamoDB client cannot be null");
        }
```

```
        if (tableName == null || tableName.trim().isEmpty()) {
            throw new IllegalArgumentException("Table name cannot be null or
empty");
        }
        if (artist == null || artist.trim().isEmpty()) {
            throw new IllegalArgumentException("Artist cannot be null or empty");
        }
        if (songTitle == null || songTitle.trim().isEmpty()) {
            throw new IllegalArgumentException("Song title cannot be null or
empty");
        }

        try {
            LOGGER.info("Adding test item to MRSC global table: " + tableName);

            Map<String, AttributeValue> item = new HashMap<>();
            item.put("Artist", AttributeValue.builder().s(artist).build());
            item.put("SongTitle", AttributeValue.builder().s(songTitle).build());

            if (album != null && !album.trim().isEmpty()) {
                item.put("Album", AttributeValue.builder().s(album).build());
            }
            if (year != null && !year.trim().isEmpty()) {
                item.put("Year", AttributeValue.builder().n(year).build());
            }

            PutItemRequest putItemRequest =
                PutItemRequest.builder().tableName(tableName).item(item).build();

            PutItemResponse response = dynamoDbClient.putItem(putItemRequest);
            LOGGER.info("Test item added successfully with strong consistency");

            return response;

        } catch (DynamoDbException e) {
            LOGGER.severe("Failed to add test item to table: " + tableName + " -
" + e.getMessage());
            throw DynamoDbException.builder()
                .message("Failed to add test item to table: " + tableName)
                .cause(e)
                .build();
        }
    }
```

Read items with consistent reads from MRSC replicas using AWS SDK for Java 2.x.

```
   public static GetItemResponse getItemWithConsistentRead(
       final DynamoDbClient dynamoDbClient, final String tableName, final String
artist, final String songTitle) {

       if (dynamoDbClient == null) {
           throw new IllegalArgumentException("DynamoDB client cannot be null");
       }
       if (tableName == null || tableName.trim().isEmpty()) {
           throw new IllegalArgumentException("Table name cannot be null or
empty");
       }
       if (artist == null || artist.trim().isEmpty()) {
           throw new IllegalArgumentException("Artist cannot be null or empty");
       }
       if (songTitle == null || songTitle.trim().isEmpty()) {
           throw new IllegalArgumentException("Song title cannot be null or
empty");
       }

       try {
           LOGGER.info("Reading item from MRSC global table with consistent
read: " + tableName);

           Map<String, AttributeValue> key = new HashMap<>();
           key.put("Artist", AttributeValue.builder().s(artist).build());
           key.put("SongTitle", AttributeValue.builder().s(songTitle).build());

           GetItemRequest getItemRequest = GetItemRequest.builder()
               .tableName(tableName)
               .key(key)
               .consistentRead(true)
               .build();

           GetItemResponse response = dynamoDbClient.getItem(getItemRequest);

           if (response.hasItem()) {
               LOGGER.info("Item found with strong consistency - no wait time
needed");
           } else {
```

```
                LOGGER.info("Item not found");
            }

            return response;

        } catch (DynamoDbException e) {
            LOGGER.severe("Failed to read item from table: " + tableName + " - "
+ e.getMessage());
            throw DynamoDbException.builder()
                .message("Failed to read item from table: " + tableName)
                .cause(e)
                .build();
        }
    }
```

Perform conditional updates with MRSC guarantees using AWS SDK for Java 2.x.

```
    public static UpdateItemResponse performConditionalUpdate(
        final DynamoDbClient dynamoDbClient,
        final String tableName,
        final String artist,
        final String songTitle,
        final String rating) {

        if (dynamoDbClient == null) {
            throw new IllegalArgumentException("DynamoDB client cannot be null");
        }
        if (tableName == null || tableName.trim().isEmpty()) {
            throw new IllegalArgumentException("Table name cannot be null or
empty");
        }
        if (artist == null || artist.trim().isEmpty()) {
            throw new IllegalArgumentException("Artist cannot be null or empty");
        }
        if (songTitle == null || songTitle.trim().isEmpty()) {
            throw new IllegalArgumentException("Song title cannot be null or
empty");
        }
        if (rating == null || rating.trim().isEmpty()) {
            throw new IllegalArgumentException("Rating cannot be null or empty");
        }
```

```
        try {
            LOGGER.info("Performing conditional update on MRSC global table: " +
tableName);

            Map<String, AttributeValue> key = new HashMap<>();
            key.put("Artist", AttributeValue.builder().s(artist).build());
            key.put("SongTitle", AttributeValue.builder().s(songTitle).build());

            Map<String, String> expressionAttributeNames = new HashMap<>();
            expressionAttributeNames.put("#rating", "Rating");

            Map<String, AttributeValue> expressionAttributeValues = new
HashMap<>();
            expressionAttributeValues.put(
                ":rating", AttributeValue.builder().n(rating).build());

            UpdateItemRequest updateItemRequest = UpdateItemRequest.builder()
                .tableName(tableName)
                .key(key)
                .updateExpression("SET #rating = :rating")
                .conditionExpression("attribute_exists(Artist)")
                .expressionAttributeNames(expressionAttributeNames)
                .expressionAttributeValues(expressionAttributeValues)
                .build();

            UpdateItemResponse response =
dynamoDbClient.updateItem(updateItemRequest);
            LOGGER.info("Conditional update successful - demonstrates strong
consistency");

            return response;

        } catch (ConditionalCheckFailedException e) {
            LOGGER.warning("Conditional check failed: " + e.getMessage());
            throw e;
        } catch (DynamoDbException e) {
            LOGGER.severe("Failed to perform conditional update: " + tableName +
" - " + e.getMessage());
            throw DynamoDbException.builder()
                .message("Failed to perform conditional update: " + tableName)
                .cause(e)
                .build();
        }
    }
```

Wait for MRSC replicas and witnesses to become active using AWS SDK for Java 2.x.

```
   public static void waitForMRSCReplicasActive(
       final DynamoDbClient dynamoDbClient, final String tableName, final int
maxWaitTimeSeconds)
       throws InterruptedException {

       if (dynamoDbClient == null) {
           throw new IllegalArgumentException("DynamoDB client cannot be null");
       }
       if (tableName == null || tableName.trim().isEmpty()) {
           throw new IllegalArgumentException("Table name cannot be null or
empty");
       }
       if (maxWaitTimeSeconds <= 0) {
           throw new IllegalArgumentException("Max wait time must be positive");
       }

       try {
           LOGGER.info("Waiting for MRSC replicas and witnesses to become
active: " + tableName);

           final long startTime = System.currentTimeMillis();
           final long maxWaitTimeMillis = maxWaitTimeSeconds * 1000L;
           int backoffSeconds = 5; // Start with 5 second intervals
           final int maxBackoffSeconds = 30; // Cap at 30 seconds

           while (System.currentTimeMillis() - startTime < maxWaitTimeMillis) {
               DescribeTableResponse response =
describeMRSCTable(dynamoDbClient, tableName);

               boolean allActive = true;
               StringBuilder statusReport = new StringBuilder();

               if (response.table().multiRegionConsistency() == null
                   || !MultiRegionConsistency.STRONG
                       .toString()

.equals(response.table().multiRegionConsistency().toString())) {
                   allActive = false;
                   statusReport
```

```
                        .append("MultiRegionConsistency: ")
                        .append(response.table().multiRegionConsistency())
                        .append(" ");
                }
                if (response.table().replicas() == null
                    || response.table().replicas().isEmpty()) {
                    allActive = false;
                    statusReport.append("No replicas found. ");
                }
                if (response.table().globalTableWitnesses() == null
                    || response.table().globalTableWitnesses().isEmpty()) {
                    allActive = false;
                    statusReport.append("No witnesses found. ");
                }

                // Check table status
                if (!"ACTIVE".equals(response.table().tableStatus().toString()))
{
                    allActive = false;
                    statusReport
                        .append("Table: ")
                        .append(response.table().tableStatus())
                        .append(" ");
                }

                // Check replica status
                if (response.table().replicas() != null) {
                    for (var replica : response.table().replicas()) {
                        if (!"ACTIVE".equals(replica.replicaStatus().toString()))
{
                            allActive = false;
                            statusReport
                                .append("Replica(")
                                .append(replica.regionName())
                                .append("): ")
                                .append(replica.replicaStatus())
                                .append(" ");
                        }
                    }
                }

                // Check witness status
                if (response.table().globalTableWitnesses() != null) {
                    for (var witness : response.table().globalTableWitnesses()) {
```

```
                        if (!"ACTIVE".equals(witness.witnessStatus().toString()))
{
                            allActive = false;
                            statusReport
                                .append("Witness(")
                                .append(witness.regionName())
                                .append("): ")
                                .append(witness.witnessStatus())
                                .append(" ");
                        }
                    }
                }

                if (allActive) {
                    LOGGER.info("All MRSC replicas and witnesses are now active:
" + tableName);
                    return;
                }

                LOGGER.info("Waiting for MRSC components to become active.
Status: " + statusReport.toString());
                LOGGER.info("Next check in " + backoffSeconds + " seconds...");

                tempWait(backoffSeconds);

                // Exponential backoff with cap
                backoffSeconds = Math.min(backoffSeconds * 2, maxBackoffSeconds);
            }

            throw DynamoDbException.builder()
                .message("Timeout waiting for MRSC replicas to become active
after " + maxWaitTimeSeconds + " seconds")
                .build();

        } catch (DynamoDbException | InterruptedException e) {
            LOGGER.severe("Failed to wait for MRSC replicas to become active: " +
tableName + " - " + e.getMessage());
            throw e;
        }
    }
```

Clean up MRSC replicas and witnesses using AWS SDK for Java 2.x.

```
   public static UpdateTableResponse cleanupMRSCReplicas(
       final DynamoDbClient dynamoDbClient,
       final String tableName,
       final Region replicaRegion,
       final Region witnessRegion) {

       if (dynamoDbClient == null) {
           throw new IllegalArgumentException("DynamoDB client cannot be null");
       }
       if (tableName == null || tableName.trim().isEmpty()) {
           throw new IllegalArgumentException("Table name cannot be null or
empty");
       }
       if (replicaRegion == null) {
           throw new IllegalArgumentException("Replica region cannot be null");
       }
       if (witnessRegion == null) {
           throw new IllegalArgumentException("Witness region cannot be null");
       }

       try {
           LOGGER.info("Cleaning up MRSC replicas and witnesses for table: " +
tableName);

           // Remove replica using ReplicationGroupUpdate
           ReplicationGroupUpdate replicaUpdate =
ReplicationGroupUpdate.builder()
               .delete(DeleteReplicationGroupMemberAction.builder()
                   .regionName(replicaRegion.id())
                   .build())
               .build();

           // Remove witness
           GlobalTableWitnessGroupUpdate witnessUpdate =
GlobalTableWitnessGroupUpdate.builder()
               .delete(DeleteGlobalTableWitnessGroupMemberAction.builder()
                   .regionName(witnessRegion.id())
                   .build())
               .build();

           UpdateTableRequest updateTableRequest = UpdateTableRequest.builder()
               .tableName(tableName)
               .replicaUpdates(List.of(replicaUpdate))
```

```
                .globalTableWitnessUpdates(List.of(witnessUpdate))
                .build();

            UpdateTableResponse response =
dynamoDbClient.updateTable(updateTableRequest);
            LOGGER.info("MRSC cleanup initiated - removing replica and witness.
Response: " + response);

            return response;

        } catch (DynamoDbException e) {
            LOGGER.severe("Failed to cleanup MRSC replicas: " + tableName + " - "
+ e.getMessage());
            throw DynamoDbException.builder()
                .message("Failed to cleanup MRSC replicas: " + tableName)
                .cause(e)
                .build();
        }
    }
```

Complete MRSC workflow demonstration using AWS SDK for Java 2.x.

```
    public static void demonstrateCompleteMRSCWorkflow(
        final DynamoDbClient primaryClient,
        final DynamoDbClient replicaClient,
        final String tableName,
        final Region replicaRegion,
        final Region witnessRegion)
        throws InterruptedException {

        if (primaryClient == null) {
            throw new IllegalArgumentException("Primary DynamoDB client cannot be
null");
        }
        if (replicaClient == null) {
            throw new IllegalArgumentException("Replica DynamoDB client cannot be
null");
        }
        if (tableName == null || tableName.trim().isEmpty()) {
            throw new IllegalArgumentException("Table name cannot be null or
empty");
        }
```

```
        if (replicaRegion == null) {
            throw new IllegalArgumentException("Replica region cannot be null");
        }
        if (witnessRegion == null) {
            throw new IllegalArgumentException("Witness region cannot be null");
        }

        try {
            LOGGER.info("=== Starting Complete MRSC Workflow Demonstration ===");

            // Step 1: Create an empty single-Region table
            LOGGER.info("Step 1: Creating empty single-Region table");
            createRegionalTable(primaryClient, tableName);

            // Use the existing GlobalTableOperations method for basic table
waiting
            LOGGER.info("Intermediate step: Waiting for table [" + tableName + "]
to become active before continuing");
            GlobalTableOperations.waitForTableActive(primaryClient, tableName);

            // Step 2: Convert to MRSC with replica and witness
            LOGGER.info("Step 2: Converting to MRSC with replica and witness");
            convertToMRSCWithWitness(primaryClient, tableName, replicaRegion,
witnessRegion);

            // Wait for MRSC conversion to complete using MRSC-specific waiter
            LOGGER.info("Waiting for MRSC conversion to complete...");
            waitForMRSCReplicasActive(primaryClient, tableName);

            LOGGER.info("Intermediate step: Waiting for table [" + tableName + "]
to become active before continuing");
            GlobalTableOperations.waitForTableActive(primaryClient, tableName);

            // Step 3: Verify MRSC configuration
            LOGGER.info("Step 3: Verifying MRSC configuration");
            describeMRSCTable(primaryClient, tableName);

            // Step 4: Test strong consistency with data operations
            LOGGER.info("Step 4: Testing strong consistency with data
operations");

            // Add test item to primary region
            putTestItem(primaryClient, tableName, "The Beatles", "Hey Jude", "The
Beatles 1967-1970", "1968");
```

```
            // Immediately read from replica region (no wait needed with MRSC)
            LOGGER.info("Reading from replica region immediately (strong
consistency):");
            GetItemResponse getResponse =
                getItemWithConsistentRead(replicaClient, tableName, "The
Beatles", "Hey Jude");

            if (getResponse.hasItem()) {
                LOGGER.info("# Strong consistency verified - item immediately
available in replica region");
            } else {
                LOGGER.warning("# Item not found in replica region");
            }

            // Test conditional update from replica region
            LOGGER.info("Testing conditional update from replica region:");
            performConditionalUpdate(replicaClient, tableName, "The Beatles",
"Hey Jude", "5");
            LOGGER.info("# Conditional update successful - demonstrates strong
consistency");

            // Step 5: Cleanup
            LOGGER.info("Step 5: Cleaning up resources");
            cleanupMRSCReplicas(primaryClient, tableName, replicaRegion,
witnessRegion);

            // Wait for cleanup to complete using basic table waiter
            LOGGER.info("Waiting for replica cleanup to complete...");
            GlobalTableOperations.waitForTableActive(primaryClient, tableName);

            // "Halt" until replica/witness cleanup is complete
            DescribeTableResponse cleanupVerification =
describeMRSCTable(primaryClient, tableName);
            int backoffSeconds = 5; // Start with 5 second intervals
            while (cleanupVerification.table().multiRegionConsistency() != null)
{
                LOGGER.info("Waiting additional time (" + backoffSeconds + "
seconds) for MRSC cleanup to complete...");
                tempWait(backoffSeconds);

                // Exponential backoff with cap
                backoffSeconds = Math.min(backoffSeconds * 2, 30);
```

```
                cleanupVerification = describeMRSCTable(primaryClient,
tableName);
            }

            // Delete the primary table
            deleteTable(primaryClient, tableName);

            LOGGER.info("=== MRSC Workflow Demonstration Complete ===");
            LOGGER.info("");
            LOGGER.info("Key benefits of Multi-Region Strong Consistency
(MRSC):");
            LOGGER.info("- Immediate consistency across all regions (no eventual
consistency delays)");
            LOGGER.info("- Simplified application logic (no need to handle
eventual consistency)");
            LOGGER.info("- Support for conditional writes and transactions across
regions");
            LOGGER.info("- Consistent read operations from any region without
waiting");

        } catch (DynamoDbException | InterruptedException e) {
            LOGGER.severe("MRSC workflow failed: " + e.getMessage());
            throw e;
        }
    }
```

- For API details, see the following topics in *AWS SDK for Java 2.x API Reference*.
  - CreateTable
  - DeleteTable
  - DescribeTable
  - GetItem
  - PutItem
  - UpdateItem
  - UpdateTable

DynamoDB global tables security

Global tables replicas are DynamoDB tables, so you use the same methods for controlling access to replicas that you do for single-Region tables, including AWS Identity and Access Management (IAM) identity policies and resource-based policies.

This topic covers how to secure DynamoDB global tables using IAM permissions and AWS Key Management Service (AWS KMS) encryption. You learn about the service-linked roles (SLR) that allow cross-Region replication and auto-scaling, the IAM permissions needed to create, update, and delete global tables , and the differences between multi-Region eventual consistency (MREC) and multi-Region strong consistency (MRSC) tables. You also learn about AWS KMS encryption keys to manage cross-Region replication securely.

Service-linked roles for global tables

DynamoDB global tables rely on service-linked roles (SLRs) to manage cross-Region replication and auto-scaling capabilities.

You only need to set up these roles once per AWS account. Once created, the same roles serve all global tables in your account. For more information about service-linked roles, see Using service-linked roles in the *IAM User Guide*.

Replication service-linked role

Amazon DynamoDB automatically creates the `AWSServiceRoleForDynamoDBReplication` service-linked role (SLR) when you create your first global table. This role manages cross-Region replication for you.

When applying resource-based policies to replicas, ensure that you don't deny any of the permissions defined in the `AWSServiceRoleForDynamoDBReplicationPolicy` to the SLR principal, as this will interrupt replication. If you deny required SLR permissions, replication to and from affected replicas will stop, and the replica table status will change to `REPLICATION_NOT_AUTHORIZED`.

- For multi-Region eventual consistency (MREC) global tables, if a replica remains in the `REPLICATION_NOT_AUTHORIZED` state for more than 20 hours, the replica is irreversibly converted to a single-Region DynamoDB table.
- For multi-Region strong consistency (MRSC) global tables, denying required permissions results in `AccessDeniedException` for write and strongly consistent read operations. If a replica

remains in the REPLICATION_NOT_AUTHORIZED state for more than seven days, the replica becomes permanently inaccessible, and write and strongly consistent read operations will continue to fail with an error. Some management operations like replica deletion will succeed.

Auto scaling service-linked role

When configuring a global table for provisioned capacity mode, auto scaling must be configured for the global table. DynamoDB auto scaling uses the AWS Application Auto Scaling service to dynamically adjust provisioned throughput capacity on your global table replicas. The Application Auto Scaling service creates a service-linked role (SLR) named AWSServiceRoleForApplicationAutoScaling_DynamoDBTable. This service-linked role is automatically created in your AWS account when you first configure auto scaling for a DynamoDB table. It allows Application Auto Scaling to managed provisioned table capacity and create CloudWatch alarms.

When applying resource-based policies to replicas, ensure that you don't deny any permissions defined in the AWSApplicationAutoscalingDynamoDBTablePolicy to the Application Auto Scaling SLR principal, as this will interrupt auto scaling functionality.

Example IAM policies for service-linked roles

An IAM policy with the following condition does not impact required permissions to the DynamoDB replication SLR and AWS Auto Scaling SLR. This condition can be added to otherwise broadly restrictive policies to avoid unintentionally interrupting replication or auto scaling.

Excluding required SLR permissions from deny policies

The following example shows how to exclude service-linked role principals from deny statements:

```
"Condition": {
    "StringNotEquals": {
        "aws:PrincipalArn": [
            "arn:aws::iam::111122223333:role/aws-service-role/
replication.dynamodb.amazonaws.com/AWSServiceRoleForDynamoDBReplication",
            "arn:aws::iam::111122223333:role/aws-service-role/dynamodb.application-
autoscaling.amazonaws.com/AWSServiceRoleForApplicationAutoScaling_DynamoDBTable"
        ]
    }
}
```

How global tables use AWS IAM

The following sections describe the required permissions for different global table operations and provide policy examples to help you configure the appropriate access for your users and applications.

> **Note**
>
> All permissions described must be applied to the specific table resource ARN in the affected Region(s). The table resource ARN follows the format `arn:aws:dynamodb:region:account-id:table/table-name`, where you need to specify your actual Region, account ID, and table name values.

Topics

- Creating global tables and adding replicas
- Updating global tables
- Deleting global tables and removing replicas

Creating global tables and adding replicas

DynamoDB global tables support two consistency modes: multi-Region eventual consistency (MREC) and multi-Region strong consistency (MRSC). MREC global tables can have multiple replicas across any number of Regions and provide eventual consistency. MRSC global tables require exactly three Regions (three replicas or two replicas and one witness) and provide strong consistency with zero recovery point objective (RPO).

The permissions required to create global tables depend on whether you're creating a global table with or without a witness.

Permissions for creating global tables

The following permissions are required both for initial global table creation and for adding replicas later. These permissions apply to both Multi-Region Eventual Consistency (MREC) and Multi-Region Strong Consistency (MRSC) global tables.

- Global tables require cross-Region replication, which DynamoDB manages through the `AWSServiceRoleForDynamoDBReplication` service-linked role (SLR). The following

permission allows DynamoDB to create this role automatically when you create a global table for the first time:

  - `iam:CreateServiceLinkedRole`

- To create a global table or add a replica using the `UpdateTable` API, you must have the following permission on the source table resource:
  - `dynamodb:UpdateTable`
- You must have the following permissions on the table resource in the Regions for the replicas to be added:
  - `dynamodb:CreateTable`
  - `dynamodb:CreateTableReplica`
  - `dynamodb:Query`
  - `dynamodb:Scan`
  - `dynamodb:UpdateItem`
  - `dynamodb:PutItem`
  - `dynamodb:GetItem`
  - `dynamodb:DeleteItem`
  - `dynamodb:BatchWriteItem`

Additional permissions for MRSC global tables using a witness

When creating a Multi-Region Strong Consistency (MRSC) global table with a witness Region, you must have the following permission on the table resource in all participating Regions (including both replica Regions and the witness Region):

- `dynamodb:CreateGlobalTableWitness`

Example IAM policies for creating global tables

Creating MREC or MRSC global table across three Regions

The following identity-based policy allows you to create an MREC or MRSC global table named "users" across three Regions, including creating the required DynamoDB replication service-linked role.

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Sid": "AllowCreatingUsersGlobalTable",
      "Effect": "Allow",
      "Action": [
        "dynamodb:CreateTable",
        "dynamodb:CreateTableReplica",
        "dynamodb:UpdateTable",
        "dynamodb:Query",
        "dynamodb:Scan",
        "dynamodb:UpdateItem",
        "dynamodb:PutItem",
        "dynamodb:GetItem",
        "dynamodb:DeleteItem",
        "dynamodb:BatchWriteItem"
      ],
      "Resource": [
        "arn:aws:dynamodb:us-east-1:123456789012:table/users",
        "arn:aws:dynamodb:us-east-2:123456789012:table/users",
        "arn:aws:dynamodb:us-west-2:123456789012:table/users"
      ]
    },
    {
      "Sid": "AllowCreatingSLR",
      "Effect": "Allow",
      "Action": [
        "iam:CreateServiceLinkedRole"
      ],
      "Resource": [
        "arn:aws:iam::123456789012:role/aws-service-role/
replication.dynamodb.amazonaws.com/AWSServiceRoleForDynamoDBReplication"
      ]
    }
  ]
}
```

Restricting MREC or MRSC global table creation to specific Regions

The following identity-based policy allows you to create DynamoDB global tables replicas across specific Regions using the aws:RequestedRegion condition key, including creating the required DynamoDB replication service-linked role.

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Sid": "AllowAddingReplicasToSourceTable",
      "Effect": "Allow",
      "Action": [
        "dynamodb:UpdateTable"
      ],
      "Resource": "*",
      "Condition": {
        "StringEquals": {
          "aws:RequestedRegion": [
            "us-east-1"
          ]
        }
      }
    },
    {
      "Sid": "AllowCreatingReplicas",
      "Effect": "Allow",
      "Action": [
        "dynamodb:CreateTable",
        "dynamodb:CreateTableReplica",
        "dynamodb:UpdateTable",
        "dynamodb:Query",
        "dynamodb:Scan",
        "dynamodb:UpdateItem",
        "dynamodb:PutItem",
        "dynamodb:GetItem",
        "dynamodb:DeleteItem",
        "dynamodb:BatchWriteItem"
      ],
      "Resource": "*",
      "Condition": {
```

```
        "StringEquals": {
          "aws:RequestedRegion": [
            "us-east-2",
            "us-west-2"
          ]
        }
      }
    },
    {
      "Sid": "AllowCreatingSLR",
      "Effect": "Allow",
      "Action": [
        "iam:CreateServiceLinkedRole"
      ],
      "Resource": [
        "arn:aws:iam::123456789012:role/aws-service-role/
replication.dynamodb.amazonaws.com/AWSServiceRoleForDynamoDBReplication"
      ]
    }
  ]
}
```

Creating MRSC global table with witness

The following identity-based policy allows you to a create a DynamoDB MRSC global table named "users" with replicas in us-east-1 and us-east-2 and a witness in us-west-2, including creating the required DynamoDB replication service-linked role.

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Sid": "AllowCreatingUsersGlobalTableWithWitness",
      "Effect": "Allow",
      "Action": [
        "dynamodb:CreateTable",
        "dynamodb:CreateTableReplica",
        "dynamodb:CreateGlobalTableWitness",
        "dynamodb:UpdateTable",
```

```
        "dynamodb:Query",
        "dynamodb:Scan",
        "dynamodb:UpdateItem",
        "dynamodb:PutItem",
        "dynamodb:GetItem",
        "dynamodb:DeleteItem",
        "dynamodb:BatchWriteItem"
      ],
      "Resource": [
        "arn:aws:dynamodb:us-east-1:123456789012:table/users",
        "arn:aws:dynamodb:us-east-2:123456789012:table/users"
      ]
    },
    {
      "Sid": "AllowCreatingSLR",
      "Effect": "Allow",
      "Action": [
        "iam:CreateServiceLinkedRole"
      ],
      "Resource": [
        "arn:aws:iam::123456789012:role/aws-service-role/
replication.dynamodb.amazonaws.com/AWSServiceRoleForDynamoDBReplication"
      ]
    }
  ]
}
```

Restricting MRSC witness creation to specific Regions

This identity-based policy allows you to create a MRSC global table with replicas restricted to specific Regions using the aws:RequestedRegion condition key and unrestricted witness creation across all Regions, including creating the required DynamoDB replication service-linked role.

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Sid": "AllowCreatingReplicas",
      "Effect": "Allow",
```

```
      "Action": [
        "dynamodb:CreateTable",
        "dynamodb:CreateTableReplica",
        "dynamodb:UpdateTable",
        "dynamodb:Query",
        "dynamodb:Scan",
        "dynamodb:UpdateItem",
        "dynamodb:PutItem",
        "dynamodb:GetItem",
        "dynamodb:DeleteItem",
        "dynamodb:BatchWriteItem"
      ],
      "Resource": "*",
      "Condition": {
        "StringEquals": {
          "aws:RequestedRegion": [
            "us-east-1",
            "us-east-2"
          ]
        }
      }
    },
    {
      "Sid": "AllowCreatingWitness",
      "Effect": "Allow",
      "Action": [
        "dynamodb:CreateGlobalTableWitness"
      ],
      "Resource": "*"
    },
    {
      "Sid": "AllowCreatingSLR",
      "Effect": "Allow",
      "Action": [
        "iam:CreateServiceLinkedRole"
      ],
      "Resource": [
        "arn:aws:iam::123456789012:role/aws-service-role/
replication.dynamodb.amazonaws.com/AWSServiceRoleForDynamoDBReplication"
      ]
    }
  ]
}
```

Updating global tables

To modify replica settings for an existing global table using the `UpdateTable` API, you need the following permission on the table resource in the Region where you're making the API call:

- `dynamodb:UpdateTable`

You can additionally update other global table configurations, such as auto scaling policies and Time to Live settings. The following permissions are required for these additional update operations:

- To update a replica auto scaling policy with the `UpdateTableReplicaAutoScaling` API, you must have the following permissions on the table resource in all Regions containing replicas:
  - `application-autoscaling:DeleteScalingPolicy`
  - `application-autoscaling:DeleteScheduledAction`
  - `application-autoscaling:DeregisterScalableTarget`
  - `application-autoscaling:DescribeScalableTargets`
  - `application-autoscaling:DescribeScalingActivities`
  - `application-autoscaling:DescribeScalingPolicies`
  - `application-autoscaling:DescribeScheduledActions`
  - `application-autoscaling:PutScalingPolicy`
  - `application-autoscaling:PutScheduledAction`
  - `application-autoscaling:RegisterScalableTarget`
- To update Time to Live settings with the `UpdateTimeToLive` API, you must have the following permission on the table resource in all Regions containing replicas:
  - `dynamodb:UpdateTimeToLive`

  Note that Time to Live (TTL) is only supported for global tables configured with Multi-Region Eventual Consistency (MREC). For more information about how global tables work with TTL, see How DynamoDB global tables work.

Deleting global tables and removing replicas

To delete a global table, you must remove all replicas. The permissions required for this operation differ depending on whether you're deleting a global table with or without a witness Region.

Permissions for deleting global tables and removing replicas

The following permissions are required both for removing individual replicas and for completely deleting global tables. Deleting a global table configuration only removes the replication relationship between tables in different Regions. It does not delete the underlying DynamoDB table in the last remaining Region. The table in the last Region continues to exist as a standard DynamoDB table with the same data and settings. These permissions apply to both Multi-Region Eventual Consistency (MREC) and Multi-Region Strong Consistency (MRSC) global tables.

- To remove replicas from a global table using the `UpdateTable` API, you need the following permission on the table resource in the Region from which you're making the API call:
  - `dynamodb:UpdateTable`
- You need the following permissions on the table resource in each Region where you're removing a replica:
  - `dynamodb:DeleteTable`
  - `dynamodb:DeleteTableReplica`

Additional permissions for MRSC global tables using a witness

To delete a multi-Region strong consistency (MRSC) global table with a witness, you must have the following permission on the table resource in all participating Regions (including both replica Regions and the witness Region):

- `dynamodb:DeleteGlobalTableWitness`

Examples IAM policies to delete a global table replicas

Deleting global table replicas

This identity-based policy allows you to delete a DynamoDB global table named "users" and its replicas across three Regions:

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
```

```
    {
      "Effect": "Allow",
      "Action": [
        "dynamodb:UpdateTable",
        "dynamodb:DeleteTable",
        "dynamodb:DeleteTableReplica"
      ],
      "Resource": [
        "arn:aws:dynamodb:us-east-1:123456789012:table/users",
        "arn:aws:dynamodb:us-east-2:123456789012:table/users",
        "arn:aws:dynamodb:us-west-2:123456789012:table/users"
      ]
    }
  ]
}
```

Deleting a MRSC global table with a witness

This identity-based policy allows you to delete the replica and the witness of a MRSC global table named "users":

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "dynamodb:UpdateTable",
        "dynamodb:DeleteTable",
        "dynamodb:DeleteTableReplica",
        "dynamodb:DeleteGlobalTableWitness"
      ],
      "Resource": [
        "arn:aws:dynamodb:us-east-1:123456789012:table/users",
        "arn:aws:dynamodb:us-east-2:123456789012:table/users"
      ]
    }
  ]
}
```

How global tables use AWS KMS

Like all DynamoDB tables, global tables replicas always encrypt data at rest using encryption keys stored in AWS Key Management Service (AWS KMS).

All replicas in a global table must be configured with the same type of KMS key (AWS owned key, AWS managed key, or Customer managed key).

> **Important**
>
> DynamoDB requires access to the replica's encryption key to delete a replica. If you want to disable or delete a customer managed key used to encrypt a replica because you are deleting the replica, you should first delete the replica, wait for the table status on one of the remaining replicas to change to `ACTIVE`, then disable or delete the key.

For a global table configured for multi-Region eventual consistency (MREC), if you disable or revoke DynamoDB's access to a customer managed key used to encrypt a replica, replication to and from the replica will stop and the replica status will change to `INACCESSIBLE_ENCRYPTION_CREDENTIALS`. If a replica in a MREC global table remains in the `INACCESSIBLE_ENCRYPTION_CREDENTIALS` state for more than 20 hours, the replica is irreversibly converted to a single-Region DynamoDB table.

For a global table configured for multi-Region strong consistency (MRSC), if you disable or revoke DynamoDB's access to a customer managed key used to encrypt a replica, replication to and from the replica will stop, attempts to perform write or strongly consistent reads to the replica will return an error, and the replica status will change to `INACCESSIBLE_ENCRYPTION_CREDENTIALS`. If a replica in a MRSC global table remains in the `INACCESSIBLE_ENCRYPTION_CREDENTIALS` state for more than seven days, depending on the specific permissions revoked the replica will be archived or become permanently inaccessible.

DynamoDB multi-account global tables

Multi-account global tables automatically replicate your DynamoDB table data across multiple AWS Regions and multiple AWS accounts to improve resiliency, isolate workloads at the account level, and apply distinct security and governance controls. Each replica table resides in a distinct AWS account, enabling fault isolation at both the Region and account level. You can also align replicas with your AWS organizational structure. Multi-account global tables provide additional isolation, governance, and security benefits compared to same-account global tables.

Multi-account global tables provide the following benefits:

- Replicate DynamoDB table data automatically across your choice of AWS accounts and Regions
- Enhance security and governance by replicating data across accounts with distinct policies, guardrails, and compliance boundaries
- Improve operational resiliency and account-level fault isolation by placing replicas in separate AWS accounts
- Align workloads by business unit or ownership when using a multi-account strategy
- Simplify cost attribution by billing each replica to its respective AWS account

For more information, see Benefits of using multiple AWS accounts. If your workloads don't require multi-account replication, or you want simpler replica management with local overrides, you can continue to use same-account global tables.

You can configure multi-account global tables with Multi-Region eventual consistency (MREC). Global tables configured for Multi-Region strong consistency (MRSC) do not support the multi-account model.

Topics

- How DynamoDB global tables work
- Tutorials: Creating multi-account global tables
- DynamoDB global tables security

How DynamoDB global tables work

Multi-account global tables extend DynamoDB global tables fully managed, serverless, multi-Region, and multi-active capabilities to span multiple AWS accounts. Multi-account global tables replicate data across AWS Regions and accounts, providing the same active-active functionality as same-account global tables. When you write to any replica, DynamoDB replicates the data to all other replicas.

Key differences from same-account global tables include:

- Multi-account replication is supported for multi-Region eventual consistency (MREC) global tables.

- You can only add replicas by starting with a single-Region table. Converting an existing same-account global table into a multi-account setup is not supported. To migrate, you must delete existing replicas to return to a single-Region table before creating a new multi-account global table.
- Each replica must reside in a separate AWS account. For a multi-account global table with *N* replicas, you must have *N* accounts.
- Multi-account global tables use unified table settings across all replicas by default. All replicas automatically share the same configuration (such as throughput mode and TTL), and unlike same-account global tables, these settings cannot be overridden per replica.
- Customers must provide replication permissions to the DynamoDB global tables service principal in their resource policies.

Multi-account global tables use the same underlying replication technology as same-account global tables. Table settings are replicated automatically across all regional replicas, and customers cannot override or customize settings per replica. This ensures consistent configuration and predictable behavior across multiple AWS accounts participating in the same global table.

Settings in DynamoDB global tables define how a table behaves and how data is replicated across Regions. These settings are configured through DynamoDB control plane APIs during table creation or when adding a new regional replica.

When creating a multi-account global table, customers must set `GlobalTableSettingsReplicationMode = ENABLED` for each regional replica. This ensures that configuration changes made in one Region propagate automatically to all other Regions that participate in the global table.

You can enable settings replication after table creation. This supports the scenario where a table is originally created as a regional table and later upgraded to a multi-account global table.

Synchronized Settings

The following table settings are always synchronized across all replicas in a multi-account global table:

> **Note**
>
> Unlike same-account global tables, multi-account global tables do not allow per-Region overrides for these settings. The only exception is that overrides for read auto-scaling policies (tables and GSIs) are allowed as they are separate external resources.

- Capacity mode (provisioned capacity or on-demand)
- Table provisioned read and write capacity
- Table read and write auto scaling
- Local Secondary Index (LSI) definition
- Global Secondary Index (GSI) definition
- GSI provisioned read and write capacity
- GSI read and write auto scaling
- Streams definition in MREC mode
- Time To Live (TTL)
- Warm Throughput
- On-demand maximum read and write throughput

Non-Synchronized Settings

The following settings are not synchronized between replicas and must be configured independently for each replica table in each Region.

- Table Class
- Server-side Encryption (SSE) type
- Point-in-time Recovery
- Server-side Encryption (SSE) KMS Key Id
- Deletion Protection
- Kinesis Data Streams (KDSD)
- Tags
- Resource Policy
- Table Cloudwatch-Contributor Insights (CCI)

- GSI Cloudwatch-Contributor Insights (CCI)

Monitoring

Global tables configured for multi-Region eventual consistency (MREC) publish the `ReplicationLatency` metric to CloudWatch. This metric tracks the elapsed time between when an item is written to a replica table, and when that item appears in another replica in the global table. `ReplicationLatency` is expressed in milliseconds and is emitted for every source and destination Region pair in a global table.

Typical `ReplicationLatency` values depends on the distance between your chosen AWS Regions, as well as other variables like workload type and throughput. For example, a source replica in the US West (N. California) (us-west-1) Region has lower `ReplicationLatency` to the US West (Oregon) (us-west-2) Region compared to the Africa (Cape Town) (af-south-1) Region.

An increasing value for `ReplicationLatency` could indicate that updates from one replica are not propagating to other replica tables in a timely manner. In this case, you can temporarily redirect your application's read and write activity to a different AWS Region.

Handling Replication Latency Issues in Multi-account Global Tables

If `ReplicationLatency` exceeds 3 hours due to customer-induced issues on a replica table, DynamoDB sends a notification requesting the customer to address the underlying problem. Common customer-induced issues that may prevent replication include:

- Removing required permissions from the replica table's resource policy
- Opting out of an AWS Region that hosts a replica of the multi-account global table
- Denying the table's AWS KMS key permissions required to decrypt data

DynamoDB sends an initial notification within 3 hours of elevated replication latency, followed by a second notification after 20 hours if the issue remains unresolved. If the problem is not corrected within the required time window, DynamoDB will automatically disassociate the replica from the global table. The affected replica will then be converted to a regional table.

Tutorials: Creating multi-account global tables

This section provides step-by-step instructions for creating DynamoDB global tables that span across multiple AWS accounts.

Create a multi-account global table using the DynamoDB console

Follow these steps to create a multi-account global table using the AWS Management Console. The following example creates a global table with replica tables in the United States.

1. Sign in to the AWS Management Console and open the DynamoDB console at https://console.aws.amazon.com/dynamodb/ for the first account (say *111122223333*).
2. For this example, choose **US East (Ohio)** from the Region selector in the navigation bar.
3. In the navigation pane on the left side of the console, choose **Tables**.
4. Choose **Create Table**.
5. On the **Create table** page:
   a. For **Table name**, enter **`MusicTable`**.
   b. For **Partition key**, enter **`Artist`**.
   c. For **Sort key**, enter **`SongTitle`**.
   d. Keep the other default settings and choose **Create table**.
6. Add the following resource policy to the table

   JSON

```
{
"Version":"2012-10-17",
"Statement": [
    {
        "Sid": "DynamoDBActionsNeededForSteadyStateReplication",
        "Effect": "Allow",
        "Action": [
            "dynamodb:ReadDataForReplication",
            "dynamodb:WriteDataForReplication",
            "dynamodb:ReplicateSettings"
        ],
        "Resource": "arn:aws:dynamodb:us-east-2:111122223333:table/
MusicTable",
        "Principal": {"Service": ["replication.dynamodb.amazonaws.com"]},
        "Condition": {
            "StringEquals": {
                "aws:SourceAccount": ["444455556666","111122223333"],
                "aws:SourceArn": [
```

```
                    "arn:aws:dynamodb:us-east-1:444455556666:table/
MusicTable",
                    "arn:aws:dynamodb:us-east-2:111122223333:table/
MusicTable"
                ]
            }
        }
    },
    {
        "Sid": "AllowTrustedAccountsToJoinThisGlobalTable",
        "Effect": "Allow",
        "Action": [
            "dynamodb:AssociateTableReplica"
        ],
        "Resource": "arn:aws:dynamodb:us-east-2:111122223333:table/
MusicTable",
        "Principal": {"AWS": ["444455556666"]}
    }
]
}
```

7. This new table serves as the first replica table in a new global table. It is the prototype for other replica tables that you add later.
8. Wait for the table to become **Active**. For the newly created table, from the **Global tables** tab, navigate to **Settings Replication** and click **Enable**.
9. Logout of this account (*111122223333* here).
10. Sign in to the AWS Management Console and open the DynamoDB console at https://console.aws.amazon.com/dynamodb/ for the second account (say *444455556666*).
11. For this example, choose **US East (N. Virginia)** from the Region selector in the navigation bar.
12. The console ensures that a table with the same name doesn't exist in the selected Region. If a table with the same name does exist, you must delete the existing table before you can create a new replica table in that Region.
13. In the drop down near **Create Table**, choose **Create from another account**
14. On the **Create table from another account** page:

    a. Add **`arn:aws:dynamodb:us-east-2:111122223333:table/MusicTable`** as the table arn for the source table.

    b. In the **Replica Table ARNs**, add the ARN of the source table again **`arn:aws:dynamodb:us-east-2:111122223333:table/MusicTable`**. If there are

multiple replicas already existing as part of a Multi Account Global Table, you must add every existing replica to the ReplicaTableARN.

c. Keep the other default settings and choose **Submit**.

15. The **Global tables** tab for the Music table (and for any other replica tables) shows that the table has been replicated in multiple Regions.
16. To test replication:

    a. You can use any of the regions where a replica exists for this table

    b. Choose **Explore table items**.

    c. Choose **Create item**.

    d. Enter **`item_1`** for **Artist** and **`Song Value 1`** for **SongTitle**.

    e. Choose **Create item**.

    f. Verify replication by switching to the other regions:

    g. Verify that the Music table contains the item you created.

Create a multi-account global table using the AWS CLI

The following examples show how to create a multi-account global table using the AWS CLI. These examples demonstrate the complete workflow for setting up cross-account replication.

CLI

Use the following AWS CLI commands to create a multi-account global table with cross-account replication.

```
# STEP 1: Setting resource policy for the table in account 111122223333

cat > /tmp/source-resource-policy.json << 'EOF'
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "DynamoDBActionsNeededForSteadyStateReplication",
            "Effect": "Allow",
            "Action": [
                "dynamodb:ReadDataForReplication",
                "dynamodb:WriteDataForReplication",
                "dynamodb:ReplicateSettings"
```

```
            ],
            "Resource": "arn:aws:dynamodb:us-east-2:111122223333:table/MusicTable",
            "Principal": {"Service": ["replication.dynamodb.amazonaws.com"]},
            "Condition": {
                "StringEquals": {
                    "aws:SourceAccount": ["444455556666","111122223333"],
                    "aws:SourceArn": [
                        "arn:aws:dynamodb:us-east-1:444455556666:table/MusicTable",
                        "arn:aws:dynamodb:us-east-2:111122223333:table/MusicTable"
                    ]
                }
            }
        },
        {
            "Sid": "AllowTrustedAccountsToJoinThisGlobalTable",
            "Effect": "Allow",
            "Action": [
                "dynamodb:AssociateTableReplica"
            ],
            "Resource": "arn:aws:dynamodb:us-east-2:111122223333:table/MusicTable",
            "Principal": {"AWS": ["444455556666"]}
        }
    ]
}
EOF

# Step 2: Create a new table (MusicTable) in US East (Ohio),
#   with DynamoDB Streams enabled (NEW_AND_OLD_IMAGES),
#   and Settings Replication ENABLED on the account 111122223333

aws dynamodb create-table \
    --table-name MusicTable \
    --attribute-definitions \
        AttributeName=Artist,AttributeType=S \
        AttributeName=SongTitle,AttributeType=S \
    --key-schema \
        AttributeName=Artist,KeyType=HASH \
        AttributeName=SongTitle,KeyType=RANGE \
    --billing-mode PAY_PER_REQUEST \
    --stream-specification StreamEnabled=true,StreamViewType=NEW_AND_OLD_IMAGES \
    --global-table-settings-replication-mode ENABLED \
    --resource-policy file:///tmp/source-resource-policy.json \
    --region us-east-2
```

```
# Step 3: Creating replica table in account 444455556666

# Resource policy for account 444455556666
cat > /tmp/dest-resource-policy.json << 'EOF'
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "DynamoDBActionsNeededForSteadyStateReplication",
            "Effect": "Allow",
            "Action": [
                "dynamodb:ReadDataForReplication",
                "dynamodb:WriteDataForReplication",
                "dynamodb:ReplicateSettings"
            ],
            "Resource": "arn:aws:dynamodb:us-east-1:444455556666:table/MusicTable",
            "Principal": {"Service": ["replication.dynamodb.amazonaws.com"]},
            "Condition": {
                "StringEquals": {
                    "aws:SourceAccount": ["444455556666","111122223333"],
                    "aws:SourceArn": [
                        "arn:aws:dynamodb:us-east-1:444455556666:table/MusicTable",
                        "arn:aws:dynamodb:us-east-2:111122223333:table/MusicTable"
                    ]
                }
            }
        }
    ]
}
EOF

# Execute the replica table creation
aws dynamodb create-table \
    --table-name MusicTable \
    --global-table-source-arn "arn:aws:dynamodb:us-east-2:111122223333:table/
MusicTable" \
    --resource-policy file:///tmp/dest-resource-policy.json \
    --global-table-settings-replication-mode ENABLED \
    --region us-east-1

# Step 4: View the list of replicas created using describe-table
aws dynamodb describe-table \
    --table-name MusicTable \
```

```
    --region us-east-2 \
    --query 'Table.
{TableName:TableName,TableStatus:TableStatus,MultiRegionConsistency:MultiRegionConsistency,R
{Region:RegionName,Status:ReplicaStatus}}'

# Step 5: To verify that replication is working, add a new item to the Music table
 in US East (Ohio)
aws dynamodb put-item \
    --table-name MusicTable \
    --item '{"Artist": {"S":"item_1"},"SongTitle": {"S":"Song Value 1"}}' \
    --region us-east-2

# Step 6: Wait for a few seconds, and then check to see whether the item has been
# successfully replicated to US East (N. Virginia) and Europe (Ireland)
aws dynamodb get-item \
    --table-name MusicTable \
    --key '{"Artist": {"S":"item_1"},"SongTitle": {"S":"Song Value 1"}}' \
    --region us-east-1

aws dynamodb get-item \
    --table-name MusicTable \
    --key '{"Artist": {"S":"item_1"},"SongTitle": {"S":"Song Value 1"}}' \
    --region us-east-2

# Step 7: Delete the replica table in US East (N. Virginia) Region
aws dynamodb delete-table \
    --table-name MusicTable \
    --region us-east-1

# Clean up: Delete the primary table
aws dynamodb delete-table \
    --table-name MusicTable \
    --region us-east-2
```

DynamoDB global tables security

Global tables replicas are DynamoDB tables, so you use the same methods for controlling access to replicas that you do for single-Region tables, including AWS Identity and Access Management (IAM) identity policies and resource-based policies. This topic covers how to secure DynamoDB multi-account global tables using IAM permissions and AWS Key Management Service (AWS KMS) encryption. You learn about the resource based policies and service-linked roles (SLR) that allow

cross-Region cross-account replication and auto-scaling, the IAM permissions needed to create, update, and delete global tables, for a multi-Region eventual consistency (MREC) tables. You also learn about AWS KMS encryption keys to manage cross-Region replication securely.

It provides detailed information about the resource-based policies and permissions required to establish cross-account and cross-region table replication. Understanding this security model is crucial for customers who need to implement secure, cross-account data replication solutions.

Service principal authorization for replication

DynamoDB's multi-account global tables use a distinct authorization approach because replication is performed across account boundaries. This is done using DynamoDB's replication service principal: `replication.dynamodb.amazonaws.com`. Each participating account must explicitly allow that principal in the replica table's resource policy, giving it permissions that can be constrained to specific replicas by source context conditions on keys like `aws:SourceAccount`, `aws:SourceArn`, etc. — see AWS global condition keys for more details. Permissions are bi-directional, which means that all replicas must explicitly grant permissions to each other before replication can be established across any particular pair of replicas.

The following service principal permissions are essential for cross-account replication:

- `dynamodb:ReadDataForReplication` grants the ability to read data for replication purposes. This permission allows changes in one replica to be read and propagated to other replicas.
- `dynamodb:WriteDataForReplication` permits the writing of replicated data to destination tables. This permission allows changes to be synchronized across all replicas in the global table.
- `dynamodb:ReplicateSettings` enables the synchronization of table settings across replicas, providing consistent configuration across all participating tables.

Each replica must give the above permissions to all other replicas and to itself — i.e. the source context conditions must include the full set of replicas that comprises the global table. These permission are verified for each new replica when it is added to a multi-account global table. This verifies that replication operations are performed only by the authorized DynamoDB service and only between the intended tables.

Service-linked roles for multi-account global tables

DynamoDB multi-account global tables replicate settings across all replicas so that each replica is set up identically with consistent throughput and provides a seamless fail-over experience.

Replication of settings is controlled through the `ReplicateSettings` permission on the service principal, but we also rely on service-linked roles (SLRs) to manage certain cross-account cross-Region replication and auto-scaling capabilities. These roles are set up only once per AWS account. Once created, the same roles serve all global tables in your account. For more information about service-linked roles, see Using service-linked roles in the IAM User Guide.

Settings management service-linked role

Amazon DynamoDB automatically creates the AWSServiceRoleForDynamoDBGlobalTableSettingsManagement service-linked role (SLR) when you create your first multi-account global table replica in the account. This role manages cross-account cross-Region replication of settings for you.

When applying resource-based policies to replicas, confirm that you do not deny any of the permissions defined in the `AWSServiceRoleForDynamoDBGlobalTableSettingsManagement` to the SLR principal, as this could interfere with settings management and may impair replication if throughput does not match across replicas or GSIs. If you deny required SLR permissions, replication to and from affected replicas may stop, and the replica table status will change to `REPLICATION_NOT_AUTHORIZED`. For multi-account global tables, if a replica remains in the `REPLICATION_NOT_AUTHORIZED` state for more than 20 hours, the replica is irreversibly converted to a single-Region DynamoDB table. The SLR has the following permissions:

- `application-autoscaling:DeleteScalingPolicy`
- `application-autoscaling:DescribeScalableTargets`
- `application-autoscaling:DescribeScalingPolicies`
- `application-autoscaling:DeregisterScalableTarget`
- `application-autoscaling:PutScalingPolicy`
- `application-autoscaling:RegisterScalableTarget`

Auto scaling service-linked role

When configuring a global table for provisioned capacity mode, auto scaling must be configured for the global table. DynamoDB auto scaling uses the AWS Application Auto Scaling service to dynamically adjust provisioned throughput capacity on your global table replicas. The Application Auto Scaling service creates a service-linked role (SLR) named AWSServiceRoleForApplicationAutoScaling_DynamoDBTable. This service-linked role is automatically created in your AWS account when you first configure auto scaling for a DynamoDB

table. It allows Application Auto Scaling to manage provisioned table capacity and create CloudWatch alarms.

When applying resource-based policies to replicas, verify that you do not deny any permissions defined in the AWSApplicationAutoscalingDynamoDBTablePolicy to the Application Auto Scaling SLR principal, as this will interrupt auto-scaling functionality.

How global tables use AWS IAM

The following sections describe the required permissions for different global table operations and provide policy examples to help you configure the appropriate access for your users and applications.

> **Note**
>
> All permissions described must be applied to the specific table resource ARN in the affected Region(s). The table resource ARN follows the format `arn:aws:dynamodb:region:account-id:table/table-name`, where you need to specify your actual Region, account ID, and table name values.

The following are the step-by-step topics we cover in the sections below:

- Creating multi-account global tables and adding replicas
- Updating a multi-account global table
- Deleting global tables and removing replicas

Creating global tables and adding replicas

Permissions for creating global tables

When a new replica is added to a regional table to form a multi-account global table or to an existing multi-account global table, the IAM principal performing the action must be authorized by all existing members. All existing members needs to give the following permission in their table policy for the replica addition to succeed:

- `dynamodb:AssociateTableReplica` - This permission allows tables to be joined into a global table setup. This is the foundational permission that enables the initial establishment of the replication relationship.

This precise control allows only authorized accounts to participate in the global table setup.

Example IAM policies for creating global tables

Example IAM policies for a 2-replica setup

The setup of multi-account global tables follows a specific authorization flow that provides secure replication. Let's examine how this works in practice by walking through a practical scenario where a customer wants to establish a global table with two replicas. The first replica (ReplicaA) resides in Account A in the ap-east-1 region, while the second replica (ReplicaB) is in Account B in the eu-south-1 region.

- In the source account (Account A), the process begins with creating the primary replica table. The account administrator must attach a resource-based policy to this table that explicitly grants necessary permissions to the destination account (Account B) to perform the association. This policy also authorizes the DynamoDB replication service to perform essential replication actions.
- The destination account (Account B) follows a similar process by attaching a corresponding resource-based policy while creating the replica and referencing the source table ARN to be used to create the replica. This policy mirrors the permissions granted by Account A, creating a trusted bi-directional relationship. Before establishing replication, DynamoDB validates these cross-account permissions to verify proper authorization is in place.

To establish this setup:

- The administrator of Account A must first attach the resource-based policy to ReplicaA. This policy explicitly grants the necessary permissions to Account B and the DynamoDB replication service.
- Similarly, the administrator of Account B must attach a matching policy to ReplicaB, with account references reversed to grant corresponding permissions to Account A, in the create table call to create replica B referencing replica A as source table.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "DynamoDBActionsNeededForSteadyStateReplication",
```

```
            "Effect": "Allow",
            "Action": [
                "dynamodb:ReadDataForReplication",
                "dynamodb:WriteDataForReplication",
                "dynamodb:ReplicateSettings"
            ],
            "Resource": "arn:aws:dynamodb:ap-east-1:111122223333:table/ReplicaA",
            "Principal": {"Service": ["replication.dynamodb.amazonaws.com"]},
            "Condition": {
                "StringEquals": {
                    "aws:SourceAccount": [ "111122223333", "444455556666" ],
                    "aws:SourceArn": [
                        "arn:aws:dynamodb:ap-east-1:111122223333:table/ReplicaA",
                        "arn:aws:dynamodb:eu-south-1:444455556666:table/ReplicaB"
                    ]
                }
            }
        },
        {
            "Sid": "AllowTrustedAccountsToJoinThisGlobalTable",
            "Effect": "Allow",
            "Action": [
                "dynamodb:AssociateTableReplica"
            ],
            "Resource": "arn:aws:dynamodb:ap-east-1:111122223333:table/ReplicaA",
            "Principal": {"AWS": ["444455556666"]}
        }
    ]
}
```

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "DynamoDBActionsNeededForSteadyStateReplication",
            "Effect": "Allow",
            "Action": [
                "dynamodb:ReadDataForReplication",
                "dynamodb:WriteDataForReplication",
                "dynamodb:ReplicateSettings"
```

```
            ],
            "Resource": "arn:aws:dynamodb:eu-south-1:444455556666:table/
ReplicaB",
            "Principal": {"Service": ["replication.dynamodb.amazonaws.com"]},
            "Condition": {
                "StringEquals": {
                    "aws:SourceAccount": [ "111122223333", "444455556666" ],
                    "aws:SourceArn": [
                        "arn:aws:dynamodb:ap-east-1:111122223333:table/ReplicaA",
                        "arn:aws:dynamodb:eu-south-1:444455556666:table/ReplicaB"
                    ]
                }
            }
        }
    ]
}
```

Example IAM policies for a 3-replica setup

In this setup, we have 3 replicas ReplicaA, ReplicaB, and ReplicaC in Account A, Account B, and Account C, respectively. Replica A is the first replica, which starts as a regional table, and then ReplicaB and ReplicaC are added to it.

- The administrator of Account A must first attach the resource-based policy to ReplicaA allowing replication with all members, and allowing the IAM principals of Account B and Account C to add replicas.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "DynamoDBActionsNeededForSteadyStateReplication",
            "Effect": "Allow",
            "Action": [
                "dynamodb:ReadDataForReplication",
                "dynamodb:WriteDataForReplication",
                "dynamodb:ReplicateSettings"
            ],
```

```
            "Resource": "arn:aws:dynamodb:ap-east-1:111122223333:table/ReplicaA",
            "Principal": {"Service": ["replication.dynamodb.amazonaws.com"]},
            "Condition": {
                "StringEquals": {
                    "aws:SourceAccount": [ "111122223333", "444455556666",
 "123456789012" ],
                    "aws:SourceArn": [
                        "arn:aws:dynamodb:ap-east-1:111122223333:table/ReplicaA",
                        "arn:aws:dynamodb:eu-south-1:444455556666:table/
ReplicaB",
                        "arn:aws:dynamodb:us-east-1:123456789012:table/ReplicaC"
                    ]
                }
            }
        },
        {
            "Sid": "AllowTrustedAccountsToJoinThisGlobalTable",
            "Effect": "Allow",
            "Action": [
                "dynamodb:AssociateTableReplica"
            ],
            "Resource": "arn:aws:dynamodb:ap-east-1:111122223333:table/ReplicaA",
            "Principal": { "AWS": [ "444455556666", "123456789012" ] }
        }
    ]
}
```

- The administrator of Account B must add a replica (Replica B) pointing to ReplicaA as a source. Replica B has the following policy allowing replication between all members, and allowing Account C to add a replica:

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "DynamoDBActionsNeededForSteadyStateReplication",
            "Effect": "Allow",
            "Action": [
```

```
                "dynamodb:ReadDataForReplication",
                "dynamodb:WriteDataForReplication",
                "dynamodb:ReplicateSettings"
            ],
            "Resource": "arn:aws:dynamodb:eu-south-1:444455556666:table/
ReplicaB",
            "Principal": {"Service": ["replication.dynamodb.amazonaws.com"]},
            "Condition": {
                "StringEquals": {
                    "aws:SourceAccount": [ "111122223333", "444455556666",
 "123456789012" ],
                    "aws:SourceArn": [
                        "arn:aws:dynamodb:ap-east-1:111122223333:table/ReplicaA",
                        "arn:aws:dynamodb:eu-south-1:444455556666:table/
ReplicaB",
                        "arn:aws:dynamodb:us-east-1:123456789012:table/ReplicaC"
                    ]
                }
            }
        },
        {
            "Sid": "AllowTrustedAccountsToJoinThisGlobalTable",
            "Effect": "Allow",
            "Action": [
                "dynamodb:AssociateTableReplica"
            ],
            "Resource": "arn:aws:dynamodb:eu-south-1:444455556666:table/
ReplicaB",
            "Principal": { "AWS": [ "123456789012" ] }
        }
    ]
}
```

- Finally, the administrator of Account C create a replica with the following policy allowing replication permissions between all members. The policy doesn't allow any further replicas to be added.

JSON

```
{
```

```
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "DynamoDBActionsNeededForSteadyStateReplication",
            "Effect": "Allow",
            "Action": [
                "dynamodb:ReadDataForReplication",
                "dynamodb:WriteDataForReplication",
                "dynamodb:ReplicateSettings"
            ],
            "Resource": "arn:aws:dynamodb:us-east-1:123456789012:table/ReplicaC",
            "Principal": {"Service": ["replication.dynamodb.amazonaws.com"]},
            "Condition": {
                "StringEquals": {
                    "aws:SourceAccount": [ "111122223333", "444455556666" ],
                    "aws:SourceArn": [
                        "arn:aws:dynamodb:ap-east-1:111122223333:table/ReplicaA",
                        "arn:aws:dynamodb:eu-south-1:444455556666:table/ReplicaB"
                    ]
                }
            }
        }
    ]
}
```

Updating a multi-account global table

To modify replica settings for an existing global table using the UpdateTable API, you need the following permission on the table resource in the Region where you're making the API call: `dynamodb:UpdateTable`

You can additionally update other global table configurations, such as auto scaling policies and Time to Live settings. The following permissions are required for these additional update operations:

To update Time to Live settings with the `UpdateTimeToLive` API, you must have the following permission on the table resource in all Regions containing replicas: `dynamodb:UpdateTimeToLive`

To update a replica auto scaling policy with the `UpdateTableReplicaAutoScaling` API, you must have the following permissions on the table resource in all Regions containing replicas:

- application-autoscaling:DeleteScalingPolicy
- application-autoscaling:DeleteScheduledAction
- application-autoscaling:DeregisterScalableTarget
- application-autoscaling:DescribeScalableTargets
- application-autoscaling:DescribeScalingActivities
- application-autoscaling:DescribeScalingPolicies
- application-autoscaling:DescribeScheduledActions
- application-autoscaling:PutScalingPolicy
- application-autoscaling:PutScheduledAction
- application-autoscaling:RegisterScalableTarget

> **Note**
>
> You need to provide dynamodb:ReplicateSettings permissions across all replica regions and accounts for the update table to succeed. If any replica does not provide permissions to replicate settings to any replica in the multi-account global table, all Update operations across all replicas will fail with AccessDeniedException till the permissions are fixed.

Deleting global tables and removing replicas

To delete a global table, you must remove all replicas. Unlike same-account Global Table, you cannot use UpdateTable to delete a replica table in a remote region and each replica must be deleted through the DeleteTable API from the account that controls it.

Permissions for deleting global tables and removing replicas

The following permissions are required both for removing individual replicas and for completely deleting global tables. Deleting a global table configuration only removes the replication relationship between tables in different Regions. It does not delete the underlying DynamoDB table in the last remaining Region. The table in the last Region continues to exist as a standard DynamoDB table with the same data and settings.

You need the following permissions on the table resource in each Region where you're removing a replica:

- `dynamodb:DeleteTable`
- `dynamodb:DeleteTableReplica`

How global tables use AWS KMS

Like all DynamoDB tables, global table replicas always encrypt data at rest using encryption keys stored in AWS Key Management Service (AWS KMS).

> **Note**
>
> Unlike same-account global table, different replicas in a multi-account global table can be configured with the different type of AWS KMS key (AWS owned key, or Customer managed key). Multi-account global tables do not support AWS Managed Keys.

Multi-account global tables that use CMKs requires each replica's keys policy to give permissions to the DynamoDB replication service principal (`replication.dynamodb.amazonaws.com`) to access the key for replication and settings management. The following permissions are required:

- `kms:Decrypt`
- `kms:ReEncrypt*`
- `kms:GenerateDataKey*`
- `kms:DescribeKey`

Important

DynamoDB requires access to the replica's encryption key to delete a replica. If you want to disable or delete a customer managed key used to encrypt a replica because you are deleting the replica, you should first delete the replica, wait for the table to be removed from the replication group by calling describe in one of the other replicas, then disable or delete the key.

If you disable or revoke DynamoDB's access to a customer managed key used to encrypt a replica, replication to and from the replica will stop and the replica status will change to `INACCESSIBLE_ENCRYPTION_CREDENTIALS`. If a replica remains in the `INACCESSIBLE_ENCRYPTION_CREDENTIALS` state for more than 20 hours, the replica is irreversibly converted to a single-Region DynamoDB table.

Example AWS KMS policy

The AWS KMS policy allows DynamoDB to access both AWS KMS keys for replication between replicas A an B. The AWS KMS keys attached to the DynamoDB replica in each account needs to be updated with the following policy:

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
      {
        "Effect": "Allow",
        "Principal": { "Service": "replication.dynamodb.amazonaws.com" },
        "Action": [
            "kms:Decrypt",
            "kms:ReEncrypt*",
            "kms:GenerateDataKey*",
            "kms:DescribeKey"
        ],
        "Resource": "*",
        "Condition": {
            "StringEquals": {
                "aws:SourceAccount": [ "111122223333", "444455556666" ],
                "aws:SourceArn": [
                    "arn:aws:dynamodb:ap-east-1:111122223333:table/ReplicaA",
                    "arn:aws:dynamodb:eu-south-1:444455556666:table/ReplicaB"
                ]
            }
        }
      }
   ]
 }
```

Understanding Amazon DynamoDB billing for global tables

This guide describes how DynamoDB billing works for global tables, identifying the components that contribute to the cost of global tables, including a practical example.

Amazon DynamoDB global tables is a fully managed, serverless, multi-Region, and multi-active database. Global tables are designed for 99.999% availability, delivering increased application

resiliency, and improved business continuity. Global tables replicate your DynamoDB tables automatically across your choice of AWS Regions so you can achieve fast, local read and write performance.

How it works

The billing model for global tables differs from single-Region DynamoDB tables. Write operations for single-Region DynamoDB tables are billed using the following units:

- Write Request Units (WRUs) for on-demand capacity mode, where one WRU is charged for each write up to 1KB
- Write Capacity Units (WCUs) for provisioned capacity mode, where one WCU provides one write per second for up to 1 KB

When you create a global table by adding a replica table to an existing single-Region table, that single-Region table becomes a replica table, which means the units used to bill for writes to the table also change. Write operations to replica tables are billed using the following units:

- Replicated Write Request Units (rWRUs) for on-demand capacity mode, where one rWRU per replica table is charged for each write up to 1KB
- Replicated Write Capacity Units (rWCUs) for provisioned capacity mode, where one WCU per replica table provides one write per second for up to 1 KB

Updates to Global Secondary Indexes (GSIs) are billed using the same units as single-Region DynamoDB tables, even if the base table for the GSI is a replica table. Update operations for GSIs are billed using the following units:

- Write Request Units (WRUs) for on-demand capacity mode, where one WRU is charged for each write up to 1KB
- Write Capacity Units (WCUs) for provisioned capacity mode, where one WCU provides one write per second for up to 1 KB

Replicated write units (rWCUs and rWRUs) are priced the same as single-Region write units (WCUs and WRUs). Cross-Region data transfer fees apply for global tables as data is replicated across Regions. Replicated write (rWCU or rWRU) charges are incurred in every Region containing a replica table for the global table.

Read operations from single-Region tables and from replica tables use the following units::

- Read Request Units (RRUs) for on-demand capacity mode, where one RRU is charged for each strongly consistent read up to 4KB
- Read Capacity Units (RCUs) for provisioned tables, where one RCU provides one strongly consistent read per second for up to 4KB

Consistency modes and billing

The replicated write units (rWCUs and rWRUs) used to bill for write operations are identical for both multi-Region strong consistency (MRSC) and multi-Region eventual consistency (MREC) modes. Global tables using multi-Region strong consistency (MRSC) mode configured with a witness don't incur replicated write unit costs (rWCUs and rWRUs), storage costs, or data transfer costs for replication to the witness.

DynamoDB global tables billing example

Let's walk through a multi-day example scenario to see how global table write request billing works in practice (note that this example only considers write requests, and does not include the table restore and cross-Region data transfer charges that would be incurred in the example):

Day 1 - Single-Region table: You have a single-Region on-demand DynamoDB table named Table_A in the us-west-2 Region. You write 100 1KB items to Table_A. For these single-Region write operations, you are charged 1 write request unit (WRU) per 1KB written. Your day 1 charges are:

- 100 WRUs in the us-west-2 Region for single-Region writes

The total request units charged on day 1: **100 WRUs**.

Day 2 - Creating a global table: You create a global table by adding a replica to Table_A in the us-east-2 Region. Table_A is now a global table with two replica tables; one in the us-west-2 Region, and one in the us-east-2 Region. You write 150 1KB items to the replica table in the us-west-2 Region. Your day 2 charges are:

- 150 rWRUs in the us-west-2 Region for replicated writes
- 150 rWRUs in the us-east-2 Region for replicated writes

The total request units charged on day 2: **300 rWRUs**.

Day 3 - Adding a Global Secondary Index: You add a global secondary index (GSI) to the replica table in the us-east-2 Region that projects all attributes from the base (replica) table. The global table automatically creates the GSI on the replica table in the us-west-2 Region for you. You write 200 new 1KB records to the replica table in the us-west-2 Region. Your day 3 charges are:

- • 200 rWRUs in the us-west-2 Region for replicated writes
- • 200 WRUs in the us-west-2 Region for GSI updates
- • 200 rWRUs in the us-east-2 Region for replicated writes
- • 200 WRUs in the us-east-2 Region for GSI updates

The total write request units charged on day 3: **400 WRUs and 400 rWRUs**.

The total write unit charges for all three days are 500WRUs (100 WRU on day 1 + 400 WRUs on day 3) and 700 rWRUs (300 rWRUs on Day2 + 400 rWRUs on Day 3).

In summary, replica table write operations are billed in replicated write units in all Regions that contain a replica table. If you have global secondary indexes, you are charged write units for updates to GSIs in all regions that contain a GSI (which in a global table is all Regions that contain a replica table).

DynamoDB global tables versions

There are two versions of DynamoDB global tables available: Global Tables version 2019.11.21 (Current) and Global tables version 2017.11.29 (Legacy). We recommend using Global Tables version 2019.11.21 (Current), as it is easier to use, supported in more Regions, and lower cost for most workloads compared to version 2017.11.29 (Legacy).

Determining the version of a global table

Determining the version using the AWS CLI

Identifying a version 2019.11.21 (Current) global table replica

To determine if a table is a global tables version 2019.11.21 (Current) replica, invoke the `describe-table` command for the table. If the output contains the `GlobalTableVersion` attribute with a value of "2019.11.21", the table is a version 2019.11.21 (Current) global table replica.

An example CLI command for `describe-table`:

```
aws dynamodb describe-table \
--table-name users \
--region us-east-2
```

The (abridged) output contains the `GlobalTableVersion` attribute with a value of "2019.11.21", so this table is a version 2019.11.21 (Current) global table replica.

```
{
    "Table": {
        "AttributeDefinitions": [
            {
                "AttributeName": "id",
                "AttributeType": "S"
            },
            {
                "AttributeName": "name",
                "AttributeType": "S"
            }
        ],
        "TableName": "users",
        ...
        "GlobalTableVersion": "2019.11.21",
        "Replicas": [
            {
                "RegionName": "us-west-2",
                "ReplicaStatus": "ACTIVE",
            }
        ],
        ...
    }
}
```

Identifying a version 2017.11.29 (Legacy) global table replica

Global tables version 2017.11.29 (Legacy) uses a dedicated set of commands for global table management. To determine if a table is a global tables version 2017.11.29 (Legacy) replica, invoke the `describe-global-table` command for the table. If you receive a successful response, the table is a version 2017.11.29 (Legacy) global table replica. If the `describe-global-table` command returns a `GlobalTableNotFoundException` error, the table is not a version 2017.11.29 (Legacy) replica.

An example CLI command for `describe-global-table`:

```
aws dynamodb describe-global-table \
--table-name users \
--region us-east-2
```

The command returns a successful response, so this table is a version 2017.11.29 (Legacy) global table replica.

```
{
    "GlobalTableDescription": {
        "ReplicationGroup": [
            {
                "RegionName": "us-west-2"
            },
            {
                "RegionName": "us-east-2"
            }
        ],
        "GlobalTableArn": "arn:aws:dynamodb::123456789012:global-table/users",
        "CreationDateTime": "2025-06-10T13:55:53.630000-04:00",
        "GlobalTableStatus": "ACTIVE",
        "GlobalTableName": "users"
    }
}
```

Determining the version using the DynamoDB Console

To identify the version of a global table replica, perform the following:

1. Open the DynamoDB console at https://console.aws.amazon.com/dynamodb/home.
2. In the navigation pane on the left side of the console, choose **Tables**.
3. Choose the table you want to identify the global tables version for.
4. Choose the **Global Tables** tab.

   The *Summary* section displays the version of global tables in use.

Differences in behavior between Legacy and Current versions

The following list describes the differences in behavior between the Legacy and Current versions of global tables.

- version 2019.11.21 (Current) consumes less write capacity for several DynamoDB operations compared to version 2017.11.29 (Legacy), and therefore, is more cost-effective for most customers. The differences for these DynamoDB operations are as follows:
  - Invoking PutItem for a 1KB item in a Region and replicating to other Regions requires 2 rWRUs per region for 2017.11.29 (Legacy), but only 1 rWRU for 2019.11.21 (Current).
  - Invoking UpdateItem for a 1KB item requires 2 rWRUs in the source Region and 1 rWRU per destination Region for 2017.11.29 (Legacy), but only 1 rWRU for both source and destination Regions for 2019.11.21 (Current).
  - Invoking DeleteItem for a 1KB item requires 1 rWRU in the source Region and 2 rWRUs per destination Region for 2017.11.29 (Legacy), but only 1 rWRU for both source or destination Region for 2019.11.21 (Current).

  The following table shows the rWRU consumption of 2017.11.29 (Legacy) and 2019.11.21 (Current) tables for a 1KB item in two Regions.

Operation	2017.11.29 (Legacy)	2019.11.21 (Current)	Savings
PutItem	4 rWRUs	2 rWRUs	50%
UpdateItem	3 rWRUs	2 rWRUs	33%
DeleteItem	3 rWRUs	2 rWRUs	33%

- version 2017.11.29 (Legacy) is available in only 11 AWS Regions. However, version 2019.11.21 (Current) is available in all the AWS Regions.
- You create version 2017.11.29 (Legacy) global tables by first creating a set of empty Regional tables, then invoking the CreateGlobalTable API to form the global table. You create version 2019.11.21 (Current) global tables by invoking the UpdateTable API to add a replica to an existing Regional table.

- version 2017.11.29 (Legacy) requires you to empty all replicas in the table before adding a replica in a new Region (including during creation). version 2019.11.21 (Current) supports you to add and remove replicas to Regions on a table that already contains data.
- version 2017.11.29 (Legacy) uses the following dedicated set of control plane APIs for managing replicas:
  - CreateGlobalTable
  - DescribeGlobalTable
  - DescribeGlobalTableSettings
  - ListGlobalTables
  - UpdateGlobalTable
  - UpdateGlobalTableSettings

  version 2019.11.21 (Current) uses the DescribeTable and UpdateTable APIs to manage replicas.
- version 2017.11.29 (Legacy) publishes two DynamoDB Streams records for each write. version 2019.11.21 (Current) only publishes one DynamoDB Streams record for each write.
- version 2017.11.29 (Legacy) populates and updates the `aws:rep:deleting`, `aws:rep:updateregion`, and `aws:rep:updatetime` attributes. version 2019.11.21 (Current) does not populate or update these attributes.
- version 2017.11.29 (Legacy) does not synchronize Using time to live (TTL) in DynamoDB settings across replicas. version 2019.11.21 (Current) synchronizes TTL settings across replicas.
- version 2017.11.29 (Legacy) does not replicate TTL deletes to other replicas. version 2019.11.21 (Current) replicates TTL deletes to all replicas.
- version 2017.11.29 (Legacy) does not synchronize auto scaling settings across replicas. version 2019.11.21 (Current) synchronizes auto scaling settings across replicas.
- version 2017.11.29 (Legacy) does not synchronize global secondary index (GSI) settings across replicas. version 2019.11.21 (Current) synchronizes GSI settings across replicas.
- version 2017.11.29 (Legacy) does not synchronize encryption at rest settings across replicas. version 2019.11.21 (Current) synchronizes encryption at rest settings across replicas.
- version 2017.11.29 (Legacy) publishes the `PendingReplicationCount` metric. version 2019.11.21 (Current) does not publish this metric.

Upgrading to the current version

Required permissions for global tables upgrade

To upgrade to version 2019.11.21 (Current), you must have `dynamodb:UpdateGlobalTableversion` permissions in all Regions with replicas. These permissions are required in addition to the permissions needed for accessing the DynamoDB console and viewing tables.

The following IAM policy grants permissions to upgrade any global table to version 2019.11.21 (Current).

```
{
    "version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "dynamodb:UpdateGlobalTableversion",
            "Resource": "*"
        }
    ]
}
```

The following IAM policy grants permissions to upgrade only the `Music` global table with replicas in two Regions to version 2019.11.21 (Current).

```
{
    "version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "dynamodb:UpdateGlobalTableversion",
            "Resource": [
                "arn:aws:dynamodb::123456789012:global-table/Music",
                "arn:aws:dynamodb:ap-southeast-1:123456789012:table/Music",
                "arn:aws:dynamodb:us-east-2:123456789012:table/Music"
            ]
        }
    ]
}
```

What to expect during the upgrade

- All global table replicas will continue to process read and write traffic while upgrading.
- The upgrade process requires between a few minutes to several hours depending on the table size and number of replicas.
- During the upgrade process, the value of TableStatus will change from `ACTIVE` to `UPDATING`. You can view the status of the table by invoking the DescribeTable API, or with the **Tables** view in the DynamoDB console.
- Auto scaling will not adjust the provisioned capacity settings for a global table while the table is being upgraded. We strongly recommend that you set the table to on-demand capacity mode during the upgrade.
- If you choose to use provisioned capacity mode with auto scaling during the upgrade, you must increase the minimum read and write throughput on your policies to accommodate any expected increases in traffic to avoid throttling during the upgrade.
- The `ReplicationLatency` metric can temporarily report latency spikes or stop reporting metric data during the upgrade process. See, the section called "ReplicationLatency", for more information.
- When the upgrade process is complete, your table status will change to `ACTIVE`.

DynamoDB Streams behavior before, during, and after upgrade

Operation	Replica Region	Behavior before upgrade	Behavior during upgrade	Behavior after upgrade
Put or Update	**Source**	Timestamp population happens using UpdateItem.	Timestamp population happens using PutItem.	No customer visible timestamp is generated.
		Two Streams records are generated. The first record contains the customer written attribute	Two Streams records are generated. The first record contains the customer written attribute	A single Streams record is generated containing the customer-writen attributes.

Operation	Replica Region	Behavior before upgrade	Behavior during upgrade	Behavior after upgrade
		s. The second record contains the `aws:rep:*` attributes.	s. The second record contains the `aws:rep:*` attributes.	
		Two rWCUs are consumed for each customer write.	Two rWCUs are consumed for each customer write.	One rWCU is consumed for each customer write.
		`ReplicationLatency` and `PendingReplicationCount` metrics are published in CloudWatch.	`ReplicationLatency` and `PendingReplicationCount` metrics are published in CloudWatch.	`ReplicationLatency` metric is published in CloudWatch.
	Destination	Replication happens using PutItem.	Replication happens using PutItem.	Replication happens using PutItem.
		A single Streams record is generated, which contains both the customer-written attributes and the `aws:rep:*` attributes.	A single Streams record is generated, which contains both the customer-written attributes and the `aws:rep:*` attributes.	A single Streams record is generated, which contains the customer-written attributes only and no replication attributes.

Operation	Replica Region	Behavior before upgrade	Behavior during upgrade	Behavior after upgrade
		One rWCU is consumed if the item exists in the destination Region. Two rWCUs are consumed if the item doesn't exist in the destination Region.	One rWCU is consumed if the item exists in the destination Region. Two rWCUs are consumed if the item doesn't exist in the destination Region.	One rWCU is consumed for each customer write.
		`ReplicationLatency` and `PendingReplicationCount` metrics are published in CloudWatch.	`ReplicationLatency` and `PendingReplicationCount` metrics are published in CloudWatch.	`ReplicationLatency` metric is published in CloudWatch.
Delete	**Source**	Delete any item with smaller timestamp using DeleteItem.	Delete any item with smaller timestamp using DeleteItem.	Delete any item with smaller timestamp using DeleteItem.

Operation	Replica Region	Behavior before upgrade	Behavior during upgrade	Behavior after upgrade
		A single Streams record is generated, which contains both the customer-written attributes and the `aws:rep:*` attributes.	A single Streams record is generated, which contains both the customer-written attributes and the `aws:rep:*` attributes.	A single Streams record is generated, which contains the customer-written attributes.
		One rWCU is consumed for each customer delete.	One rWCU is consumed for each customer delete.	One rWCU is consumed for each customer delete.
		`ReplicationLatency` and `PendingReplicationCount` metrics are published in CloudWatch.	`ReplicationLatency` and `PendingReplicationCount` metrics are published in CloudWatch.	`ReplicationLatency` metric is published in CloudWatch.

Operation	Replica Region	Behavior before upgrade	Behavior during upgrade	Behavior after upgrade
	Destination	Two-phase deletes take place: • In Phase 1, UpdateItem sets the deleting flag. • In Phase 2, DeleteItem deletes the item.	Deletes the item using DeleteItem.	Deletes the item using DeleteItem.
		Two Streams records are generated. The first record contains the change to the `aws:rep:deleting` field. The second record contains the customer-written attributes and the `aws:rep:*` attributes.	A single Stream record is generated, which contains the customer-written attributes.	A single Stream record is generated, which contains the customer-written attributes.
		Two rWCUs are consumed for each customer delete.	One rWCU is consumed for each customer delete.	One rWCU is consumed for each customer delete.

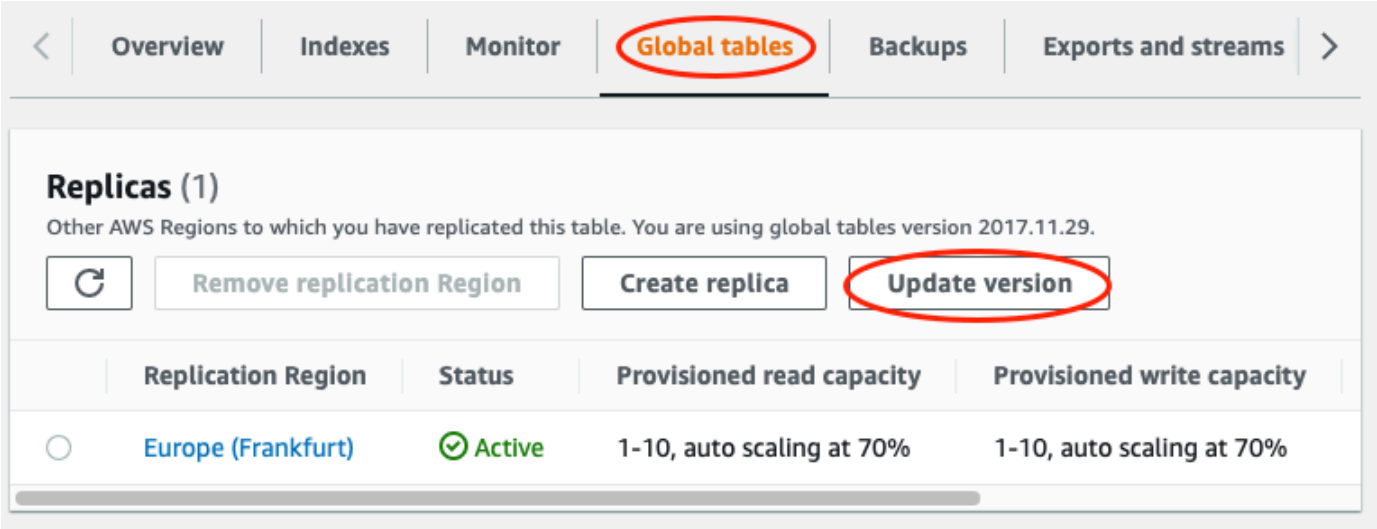
Operation	Replica Region	Behavior before upgrade	Behavior during upgrade	Behavior after upgrade
		`ReplicationLatency` and `PendingReplicationCount` metrics are published in CloudWatch.	`ReplicationLatency` metric is published in CloudWatch.	`ReplicationLatency` metric is published in CloudWatch.

Upgrading to version 2019.11.21 (Current)

Perform the following steps to upgrade your version of DynamoDB global tables using the AWS Management Console.

To upgrade global tables to version 2019.11.21 (Current)

1. Open the DynamoDB console at https://console.aws.amazon.com/dynamodb/home.
2. In the navigation pane on the left side of the console, choose **Tables**, and then select the global table that you want to upgrade to version 2019.11.21 (Current).
3. Choose the **Global Tables** tab.
4. Choose **Update version**.



5. Read and agree to the new requirements, and then choose **Update version**.

6. After the upgrade process is complete, the global tables version that appears on the console changes to **2019.11.21**.

Best practices for global tables

The following sections describe best practices for deploying and using global tables.

Version

There are two versions of DynamoDB global tables available: version 2019.11.21 (Current) and version 2017.11.29 (Legacy). You should use version 2019.11.21 (Current) whenever possible.

Deletion protection

You should enable deletion protection on global table replicas you want protected against accidental deletion. You must enable deletion protection on each replica.

Using AWS CloudFormation

CloudFormation does not currently support the coordination of multi-Region resources like global tables across stacks. If you define each replica of a global table in a separate Regional stack, you will encounter errors due to detected drift across stacks when performing replica updates. To avoid this issue, you should choose one Region as the reference Region for deploying your global tables and define all of your global table's replicas in that Region's stack.

> **Important**
>
> You cannot convert a resource of type `AWS::DynamoDB::Table` into a resource of type `AWS::DynamoDB::GlobalTable` by changing its type in your template. Attempting to convert a single-Region table to a global table by changing its CloudFormation resource type may result in the deletion of your DynamoDB table.

You can use the `AWS::DynamoDB::GlobalTable` resource to create a table in a single Region. This table will be deployed like any other single-Region table. If you later update the stack to add other Regions to a resource, replicas will be added to the table and it will safely be converted to a global table.

If you have an existing `AWS::DynamoDB::Table` resource you want to convert to a `AWS::DynamoDB::GlobalTable` resource, the recommended steps to convert the resource type are:

1. Set the `AWS::DynamoDB::Table` deletion policy to retain.
2. Remove the table from the stack definition.
3. Add replicas to the single-Region table in the AWS console, converting it to a global table.
4. Import the new global table as a new `AWS::DynamoDB::GlobalTable` resource to the stack.

Backups and Point-in-Time Recovery

Enabling automated backups and Point-in-Time Recovery (PITR) for one replica in a global table may be sufficient to meet your disaster recovery objectives. Replica backups created with AWS-Backup can be automatically replicated across Regions for greater resilience. Consider your disaster recovery plan goals in the context of multi-Region high availability when choosing your backup and PITR enablement strategy.

Designing for multi-Region high availability

For prescriptive guidance on deploying global tables, see Best Practices for DynamoDB global table design.

Working with items and attributes in DynamoDB

In Amazon DynamoDB, an *item* is a collection of attributes. Each attribute has a name and a value. An attribute value can be a scalar, a set, or a document type. For more information, see Amazon DynamoDB: How it works.

DynamoDB provides four operations for basic create, read, update, and delete (CRUD) functionality. All these operations are atomic.

- `PutItem` — Create an item.
- `GetItem` — Read an item.
- `UpdateItem` — Update an item.
- `DeleteItem` — Delete an item.

Each of these operations requires that you specify the primary key of the item that you want to work with. For example, to read an item using `GetItem`, you must specify the partition key and sort key (if applicable) for that item.

In addition to the four basic CRUD operations, DynamoDB also provides the following:

- `BatchGetItem` — Read up to 100 items from one or more tables.
- `BatchWriteItem` — Create or delete up to 25 items in one or more tables.

These batch operations combine multiple CRUD operations into a single request. In addition, the batch operations read and write items in parallel to minimize response latencies.

This section describes how to use these operations and includes related topics, such as conditional updates and atomic counters. This section also includes example code that uses the AWS SDKs.

Topics

- DynamoDB item sizes and formats
- Reading an item
- Writing an item
- Return values
- Batch operations
- Atomic counters
- Conditional writes
- Using expressions in DynamoDB
- Using time to live (TTL) in DynamoDB
- Querying tables in DynamoDB
- Scanning tables in DynamoDB
- PartiQL - a SQL-compatible query language for Amazon DynamoDB
- Working with items: Java
- Working with items: .NET

DynamoDB item sizes and formats

DynamoDB tables are schemaless, except for the primary key, so the items in a table can all have different attributes, sizes, and data types.

The total size of an item is the sum of the lengths of its attribute names and values, plus any applicable overhead as described below. You can use the following guidelines to estimate attribute sizes:

- Strings are Unicode with UTF-8 binary encoding. The size of a string is *(number of UTF-8-encoded bytes of attribute name) + (number of UTF-8-encoded bytes)*.
- Numbers are variable length, with up to 38 significant digits. Leading and trailing zeroes are trimmed. The size of a number is approximately *(number of UTF-8-encoded bytes of attribute name) + (1 byte per two significant digits) + (1 byte)*.
- A binary value must be encoded in base64 format before it can be sent to DynamoDB, but the value's raw byte length is used for calculating size. The size of a binary attribute is *(number of UTF-8-encoded bytes of attribute name) + (number of raw bytes)*.
- The size of a null attribute or a Boolean attribute is *(number of UTF-8-encoded bytes of attribute name) + (1 byte)*.
- An attribute of type `List` or `Map` requires 3 bytes of overhead, regardless of its contents. The size of a `List` or `Map` is *(number of UTF-8-encoded bytes of attribute name) + sum (size of nested elements) + (3 bytes)* . The size of an empty `List` or `Map` is *(number of UTF-8-encoded bytes of attribute name) + (3 bytes)*.
- Each `List` or `Map` element also requires 1 byte of overhead.

> **Note**
>
> We recommend that you choose shorter attribute names rather than long ones. This helps you reduce the amount of storage required, but also can lower the amount of RCU/WCUs you use.

For storage billing purposes, each item includes a per-item storage overhead that depends on the features you have enabled.

- All items in DynamoDB require 100 bytes of storage overhead for indexing.

- Some DynamoDB features (global tables, transactions, change data capture for Kinesis Data Streams with DynamoDB) require additional storage overhead to account for system-created attributes resulting from enabling those features. For example, global tables requires an additional 48 bytes of storage overhead.

Reading an item

To read an item from a DynamoDB table, use the `GetItem` operation. You must provide the name of the table, along with the primary key of the item you want.

Example

The following AWS CLI example shows how to read an item from the `ProductCatalog` table.

```
aws dynamodb get-item \
    --table-name ProductCatalog \
    --key '{"Id":{"N":"1"}}'
```

> **Note**
>
> With `GetItem`, you must specify the *entire* primary key, not just part of it. For example, if a table has a composite primary key (partition key and sort key), you must supply a value for the partition key and a value for the sort key.

A `GetItem` request performs an eventually consistent read by default. You can use the `ConsistentRead` parameter to request a strongly consistent read instead. (This consumes additional read capacity units, but it returns the most up-to-date version of the item.)

`GetItem` returns all of the item's attributes. You can use a *projection expression* to return only some of the attributes. For more information, see Using projection expressions in DynamoDB.

To return the number of read capacity units consumed by `GetItem`, set the `ReturnConsumedCapacity` parameter to `TOTAL`.

Example

The following AWS Command Line Interface (AWS CLI) example shows some of the optional `GetItem` parameters.

```
aws dynamodb get-item \
    --table-name ProductCatalog \
    --key '{"Id":{"N":"1"}}' \
    --consistent-read \
    --projection-expression "Description, Price, RelatedItems" \
    --return-consumed-capacity TOTAL
```

Writing an item

To create, update, or delete an item in a DynamoDB table, use one of the following operations:

- `PutItem`
- `UpdateItem`
- `DeleteItem`

For each of these operations, you must specify the entire primary key, not just part of it. For example, if a table has a composite primary key (partition key and sort key), you must provide a value for the partition key and a value for the sort key.

To return the number of write capacity units consumed by any of these operations, set the `ReturnConsumedCapacity` parameter to one of the following:

- `TOTAL` — Returns the total number of write capacity units consumed.
- `INDEXES` — Returns the total number of write capacity units consumed, with subtotals for the table and any secondary indexes that were affected by the operation.
- `NONE` — No write capacity details are returned. (This is the default.)

PutItem

`PutItem` creates a new item. If an item with the same key already exists in the table, it is replaced with the new item.

Example

Write a new item to the `Thread` table. The primary key for `Thread` consists of `ForumName` (partition key) and `Subject` (sort key).

```
aws dynamodb put-item \
```

```
    --table-name Thread \
    --item file://item.json
```

The arguments for `--item` are stored in the `item.json` file.

```
{
    "ForumName": {"S": "Amazon DynamoDB"},
    "Subject": {"S": "New discussion thread"},
    "Message": {"S": "First post in this thread"},
    "LastPostedBy": {"S": "fred@example.com"},
    "LastPostDateTime": {"S": "201603190422"}
}
```

UpdateItem

If an item with the specified key does not exist, `UpdateItem` creates a new item. Otherwise, it modifies an existing item's attributes.

You use an *update expression* to specify the attributes that you want to modify and their new values. For more information, see Using update expressions in DynamoDB.

Within the update expression, you use expression attribute values as placeholders for the actual values. For more information, see Using expression attribute values in DynamoDB.

Example

Modify various attributes in the `Thread` item. The optional `ReturnValues` parameter shows the item as it appears after the update. For more information, see Return values.

```
aws dynamodb update-item \
    --table-name Thread \
    --key file://key.json \
    --update-expression "SET Answered = :zero, Replies = :zero, LastPostedBy
 = :lastpostedby" \
    --expression-attribute-values file://expression-attribute-values.json \
    --return-values ALL_NEW
```

The arguments for `--key` are stored in the `key.json` file.

```
{
```

```
    "ForumName": {"S": "Amazon DynamoDB"},
    "Subject": {"S": "New discussion thread"}
}
```

The arguments for `--expression-attribute-values` are stored in the `expression-attribute-values.json` file.

```
{
    ":zero": {"N":"0"},
    ":lastpostedby": {"S":"barney@example.com"}
}
```

DeleteItem

`DeleteItem` deletes the item with the specified key.

Example

The following AWS CLI example shows how to delete the `Thread` item.

```
aws dynamodb delete-item \
    --table-name Thread \
    --key file://key.json
```

Return values

In some cases, you might want DynamoDB to return certain attribute values as they appeared before or after you modified them. The `PutItem`, `UpdateItem`, and `DeleteItem` operations have a `ReturnValues` parameter that you can use to return the attribute values before or after they are modified.

The default value for `ReturnValues` is `NONE`, meaning that DynamoDB does not return any information about attributes that were modified.

The following are the other valid settings for `ReturnValues`, organized by DynamoDB API operation.

PutItem

- `ReturnValues`: `ALL_OLD`

- If you overwrite an existing item, `ALL_OLD` returns the entire item as it appeared before the overwrite.
- If you write a nonexistent item, `ALL_OLD` has no effect.

UpdateItem

The most common usage for `UpdateItem` is to update an existing item. However, `UpdateItem` actually performs an *upsert*, meaning that it automatically creates the item if it doesn't already exist.

- `ReturnValues`: `ALL_OLD`
  - If you update an existing item, `ALL_OLD` returns the entire item as it appeared before the update.
  - If you update a nonexistent item (upsert), `ALL_OLD` has no effect.
- `ReturnValues`: `ALL_NEW`
  - If you update an existing item, `ALL_NEW` returns the entire item as it appeared after the update.
  - If you update a nonexistent item (upsert), `ALL_NEW` returns the entire item.
- `ReturnValues`: `UPDATED_OLD`
  - If you update an existing item, `UPDATED_OLD` returns only the updated attributes, as they appeared before the update.
  - If you update a nonexistent item (upsert), `UPDATED_OLD` has no effect.
- `ReturnValues`: `UPDATED_NEW`
  - If you update an existing item, `UPDATED_NEW` returns only the affected attributes, as they appeared after the update.
  - If you update a nonexistent item (upsert), `UPDATED_NEW` returns only the updated attributes, as they appear after the update.

DeleteItem

- `ReturnValues`: `ALL_OLD`
  - If you delete an existing item, `ALL_OLD` returns the entire item as it appeared before you deleted it.

- If you delete a nonexistent item, `ALL_OLD` doesn't return any data.

Batch operations

For applications that need to read or write multiple items, DynamoDB provides the `BatchGetItem` and `BatchWriteItem` operations. Using these operations can reduce the number of network round trips from your application to DynamoDB. In addition, DynamoDB performs the individual read or write operations in parallel. Your applications benefit from this parallelism without having to manage concurrency or threading.

The batch operations are essentially wrappers around multiple read or write requests. For example, if a `BatchGetItem` request contains five items, DynamoDB performs five `GetItem` operations on your behalf. Similarly, if a `BatchWriteItem` request contains two put requests and four delete requests, DynamoDB performs two `PutItem` and four `DeleteItem` requests.

In general, a batch operation does not fail unless *all* the requests in the batch fail. For example, suppose that you perform a `BatchGetItem` operation, but one of the individual `GetItem` requests in the batch fails. In this case, `BatchGetItem` returns the keys and data from the `GetItem` request that failed. The other `GetItem` requests in the batch are not affected.

BatchGetItem

A single `BatchGetItem` operation can contain up to 100 individual `GetItem` requests and can retrieve up to 16 MB of data. In addition, a `BatchGetItem` operation can retrieve items from multiple tables.

Example

Retrieve two items from the `Thread` table, using a projection expression to return only some of the attributes.

```
aws dynamodb batch-get-item \
    --request-items file://request-items.json
```

The arguments for `--request-items` are stored in the `request-items.json` file.

```
{
    "Thread": {
        "Keys": [
```

```
            {
                "ForumName":{"S": "Amazon DynamoDB"},
                "Subject":{"S": "DynamoDB Thread 1"}
            },
            {
                "ForumName":{"S": "Amazon S3"},
                "Subject":{"S": "S3 Thread 1"}
            }
        ],
        "ProjectionExpression":"ForumName, Subject, LastPostedDateTime, Replies"
    }
}
```

BatchWriteItem

The `BatchWriteItem` operation can contain up to 25 individual `PutItem` and `DeleteItem` requests and can write up to 16 MB of data. (The maximum size of an individual item is 400 KB.) In addition, a `BatchWriteItem` operation can put or delete items in multiple tables.

> **Note**
>
> `BatchWriteItem` does not support `UpdateItem` requests.

Example

Write two items to the `ProductCatalog` table.

```
aws dynamodb batch-write-item \
    --request-items file://request-items.json
```

The arguments for `--request-items` are stored in the `request-items.json` file.

```
{
    "ProductCatalog": [
        {
            "PutRequest": {
                "Item": {
                    "Id": { "N": "601" },
                    "Description": { "S": "Snowboard" },
                    "QuantityOnHand": { "N": "5" },
```

```
                    "Price": { "N": "100" }
                }
            }
        },
        {
            "PutRequest": {
                "Item": {
                    "Id": { "N": "602" },
                    "Description": { "S": "Snow shovel" }
                }
            }
        }
    ]
}
```

Atomic counters

You can use the `UpdateItem` operation to implement an *atomic counter*—a numeric attribute that is incremented, unconditionally, without interfering with other write requests. (All write requests are applied in the order in which they were received.) With an atomic counter, the updates are not idempotent. In other words, the numeric value increments or decrements each time you call `UpdateItem`. If the increment value used to update the atomic counter is positive, then it can cause overcounting. If the increment value is negative, then it can cause undercounting.

You might use an atomic counter to track the number of visitors to a website. In this case, your application would increment a numeric value, regardless of its current value. If an `UpdateItem` operation fails, the application could simply retry the operation. This would risk updating the counter twice, but you could probably tolerate a slight overcounting or undercounting of website visitors.

An atomic counter would not be appropriate where overcounting or undercounting can't be tolerated (for example, in a banking application). In this case, it is safer to use a conditional update instead of an atomic counter.

For more information, see Incrementing and decrementing numeric attributes.

Example

The following AWS CLI example increments the `Price` of a product by 5. For this example, the item was known to exist before the counter is updated. Because `UpdateItem` is not idempotent, the `Price` increases every time you run this code.

```
aws dynamodb update-item \
    --table-name ProductCatalog \
    --key '{"Id": { "N": "601" }}' \
    --update-expression "SET Price = Price + :incr" \
    --expression-attribute-values '{":incr":{"N":"5"}}' \
    --return-values UPDATED_NEW
```

Conditional writes

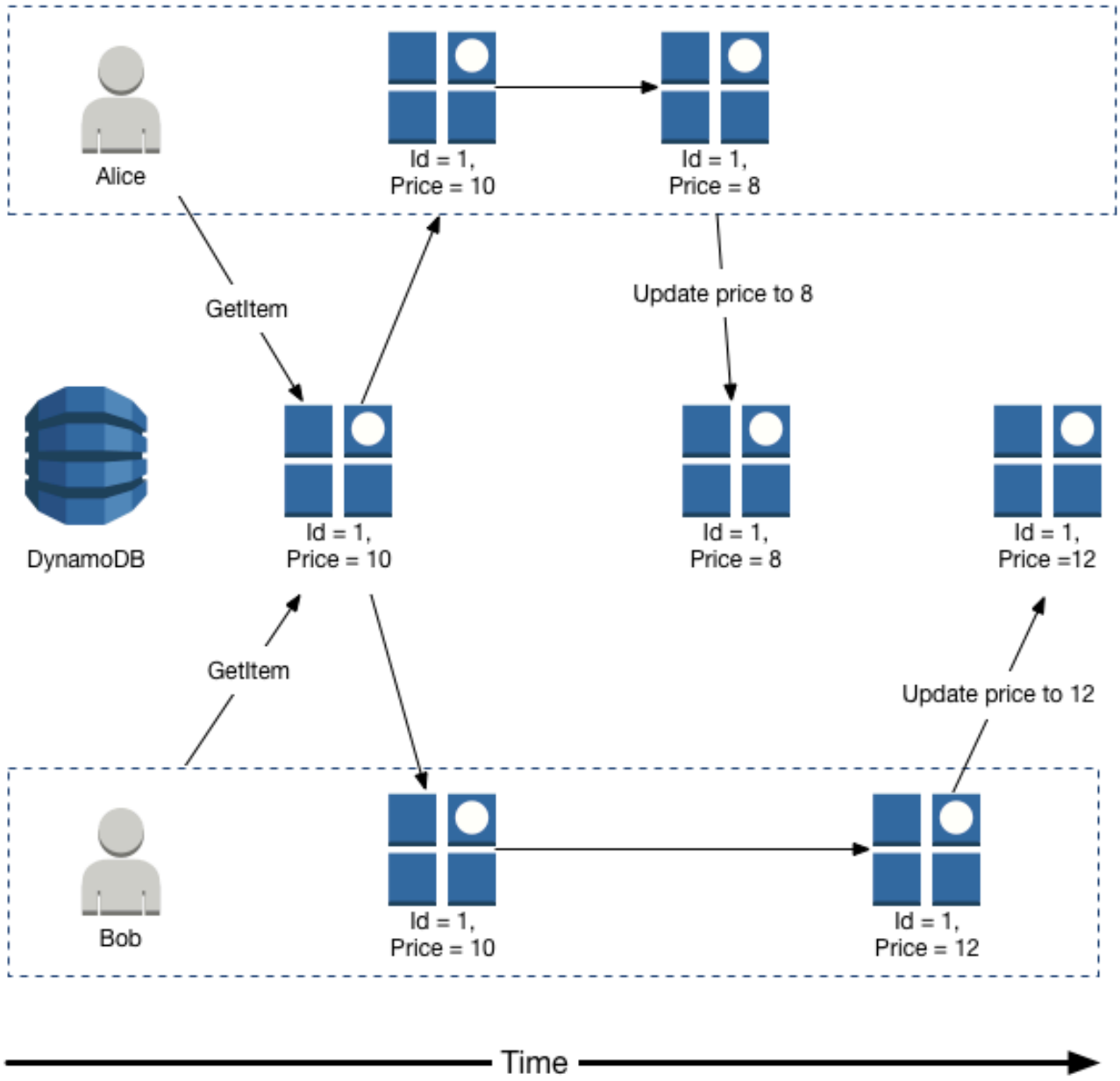
By default, the DynamoDB write operations (`PutItem`, `DeleteItem`) are *unconditional*: Each operation overwrites an existing item that has the specified primary key.

DynamoDB optionally supports conditional writes for these operations. A conditional write succeeds only if the item attributes meet one or more expected conditions. Otherwise, it returns an error.

Conditional writes check their conditions against the most recently updated version of the item. Note that if the item did not previously exist or if the most recent successful operation against that item was a delete, then the conditional write will find no previous item.

Conditional writes are helpful in many situations. For example, you might want a `PutItem` operation to succeed only if there is not already an item with the same primary key. Or you could prevent an `UpdateItem` operation from modifying an item if one of its attributes has a certain value.

Conditional writes are helpful in cases where multiple users attempt to modify the same item. Consider the following diagram, in which two users (Alice and Bob) are working with the same item from a DynamoDB table.



Suppose that Alice uses the AWS CLI to update the `Price` attribute to 8.

```
aws dynamodb update-item \
    --table-name ProductCatalog \
    --key '{"Id":{"N":"1"}}' \
    --update-expression "SET Price = :newval" \
```

```
    --expression-attribute-values file://expression-attribute-values.json
```

The arguments for `--expression-attribute-values` are stored in the file `expression-attribute-values.json`:

```
{
    ":newval":{"N":"8"}
}
```

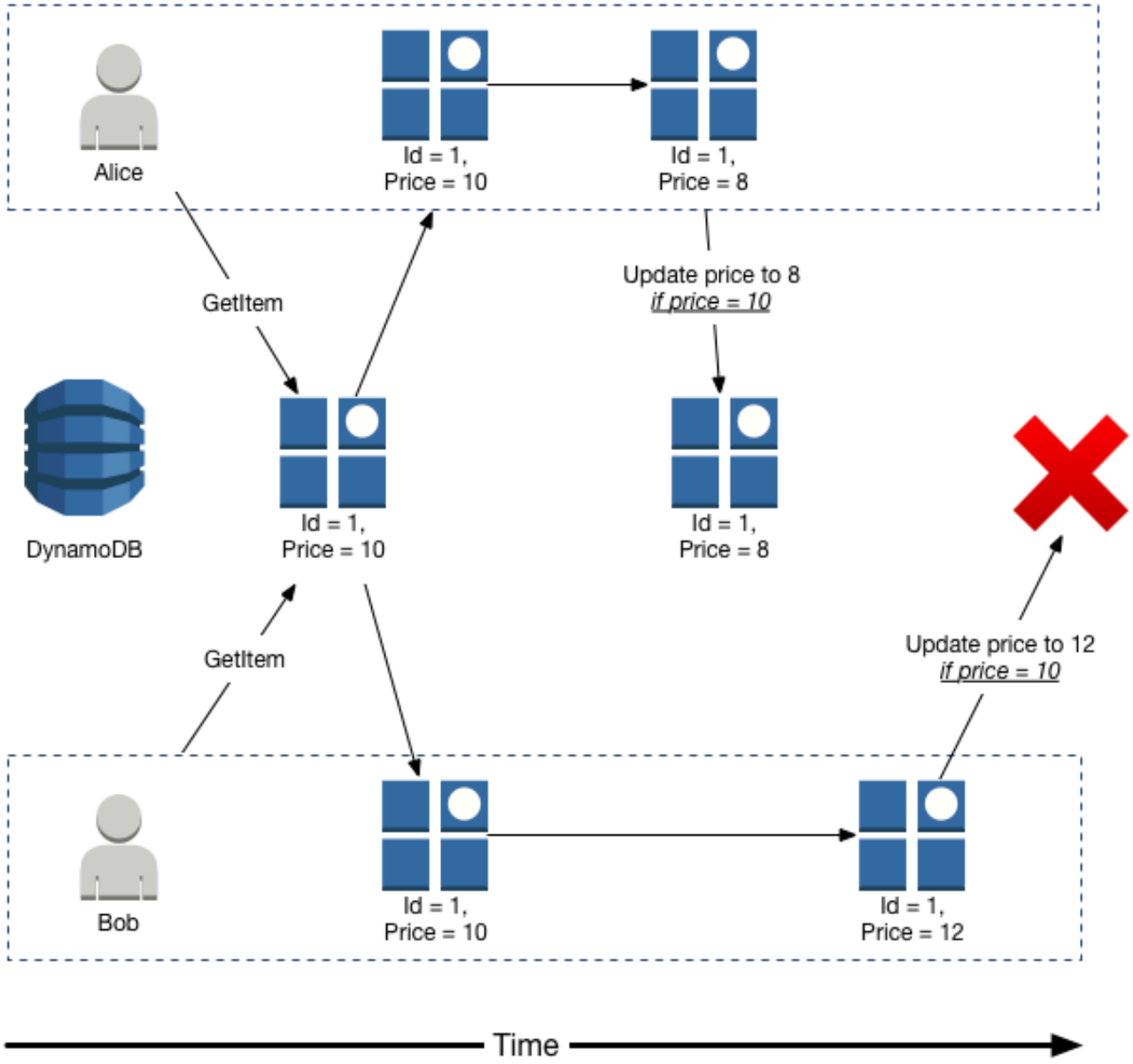
Now suppose that Bob issues a similar `UpdateItem` request later, but changes the `Price` to 12. For Bob, the `--expression-attribute-values` parameter looks like the following.

```
{
    ":newval":{"N":"12"}
}
```

Bob's request succeeds, but Alice's earlier update is lost.

To request a conditional `PutItem`, `DeleteItem`, or `UpdateItem`, you specify a condition expression. A *condition expression* is a string containing attribute names, conditional operators, and built-in functions. The entire expression must evaluate to true. Otherwise, the operation fails.

Now consider the following diagram, showing how conditional writes would prevent Alice's update from being overwritten.



Alice first tries to update `Price` to 8, but only if the current `Price` is 10.

```
aws dynamodb update-item \
    --table-name ProductCatalog \
    --key '{"Id":{"N":"1"}}' \
    --update-expression "SET Price = :newval" \
    --condition-expression "Price = :currval" \
```

```
    --expression-attribute-values file://expression-attribute-values.json
```

The arguments for `--expression-attribute-values` are stored in the `expression-attribute-values.json` file.

```
{
    ":newval":{"N":"8"},
    ":currval":{"N":"10"}
}
```

Alice's update succeeds because the condition evaluates to true.

Next, Bob attempts to update the `Price` to 12, but only if the current `Price` is 10. For Bob, the `--expression-attribute-values` parameter looks like the following.

```
{
    ":newval":{"N":"12"},
    ":currval":{"N":"10"}
}
```

Because Alice has previously changed the `Price` to 8, the condition expression evaluates to false, and Bob's update fails.

For more information, see DynamoDB condition expression CLI example.

Conditional write idempotence

Conditional writes can be *idempotent* if the conditional check is on the same attribute that is being updated. This means that DynamoDB performs a given write request only if certain attribute values in the item match what you expect them to be at the time of the request.

For example, suppose that you issue an `UpdateItem` request to increase the `Price` of an item by 3, but only if the `Price` is currently 20. After you send the request, but before you get the results back, a network error occurs, and you don't know whether the request was successful. Because this conditional write is idempotent, you can retry the same `UpdateItem` request, and DynamoDB updates the item only if the `Price` is currently 20.

Capacity units consumed by conditional writes

If a `ConditionExpression` evaluates to false during a conditional write, DynamoDB still consumes write capacity from the table. The amount consumed is dependent on the size of the

existing item (or a minimum of 1). For example, if an existing item is 300kb and the new item you are trying to create or update is 310kb, the write capacity units consumed will be the 300 if the condition fails, and 310 if the condition succeeds. If this is a new item (no existing item), then the write capacity units consumed will be 1 if the condition fails and 310 if the condition succeeds.

> **Note**
>
> Write operations consume *write* capacity units only. They never consume *read* capacity units.

A failed conditional write returns a `ConditionalCheckFailedException`. When this occurs, you don't receive any information in the response about the write capacity that was consumed. .

To return the number of write capacity units consumed during a conditional write, you use the `ReturnConsumedCapacity` parameter:

- `TOTAL` — Returns the total number of write capacity units consumed.
- `INDEXES` — Returns the total number of write capacity units consumed, with subtotals for the table and any secondary indexes that were affected by the operation.
- `NONE` — No write capacity details are returned. (This is the default.)

> **Note**
>
> Unlike a global secondary index, a local secondary index shares its provisioned throughput capacity with its table. Read and write activity on a local secondary index consumes provisioned throughput capacity from the table.

Using expressions in DynamoDB

In Amazon DynamoDB, you can use *expressions* to specify which attributes to read from an item, write data when a condition is met, specify how to update an item, define queries and filter the results of a query.

This table describes the basic expression grammar and the available kinds of expressions.

Expression type	Description
Projection expression	A projection expression identifies the attributes that you want to retrieve from an item when you use operations such as GetItem, Query, or Scan.
Condition expression	A condition expression determines which items should be modified when you use the PutItem, UpdateItem, and DeleteItem operations.
Update expression	An update expression specifies how UpdateItem will modify the attributes of an item— for example, setting a scalar value or removing elements from a list or a map.
Key condition expression	A key condition expression determines which items a query will read from a table or index.
Filter expression	A filter expression determines which items among the Query results should be returned to you. All the other results are discarded.

For information about expression syntax and more detailed information about each type of expression, see the following sections.

Topics

- Referring to item attributes when using expressions in DynamoDB
- Expression attribute names (aliases) in DynamoDB
- Using expression attribute values in DynamoDB
- Using projection expressions in DynamoDB
- Using update expressions in DynamoDB
- Condition and filter expressions, operators, and functions in DynamoDB
- DynamoDB condition expression CLI example

> **Note**
>
> For backward compatibility, DynamoDB also supports conditional parameters that do not use expressions. For more information, see Legacy DynamoDB conditional parameters. New applications should use expressions rather than the legacy parameters.

Referring to item attributes when using expressions in DynamoDB

This section describes how to refer to item attributes in an expression in Amazon DynamoDB. You can work with any attribute, even if it is deeply nested within multiple lists and maps.

Topics

- Top-level attributes
- Nested attributes
- Document paths

A Sample Item: ProductCatalog

The examples on this page use the following sample item in the `ProductCatalog` table. (This table is described in Example tables and data for use in DynamoDB.)

```
{
    "Id": 123,
    "Title": "Bicycle 123",
    "Description": "123 description",
    "BicycleType": "Hybrid",
    "Brand": "Brand-Company C",
    "Price": 500,
    "Color": ["Red", "Black"],
    "ProductCategory": "Bicycle",
    "InStock": true,
    "QuantityOnHand": null,
    "RelatedItems": [
        341,
        472,
        649
    ],
    "Pictures": {
        "FrontView": "http://example.com/products/123_front.jpg",
```

```
        "RearView": "http://example.com/products/123_rear.jpg",
        "SideView": "http://example.com/products/123_left_side.jpg"
    },
    "ProductReviews": {
     "FiveStar": [
       "Excellent! Can't recommend it highly enough! Buy it!",
       "Do yourself a favor and buy this."
     ],
     "OneStar": [
       "Terrible product! Do not buy this."
     ]
    },
    "Comment": "This product sells out quickly during the summer",
    "Safety.Warning": "Always wear a helmet"
}
```

Note the following:

- The partition key value (`Id`) is `123`. There is no sort key.
- Most of the attributes have scalar data types, such as `String`, `Number`, `Boolean`, and `Null`.
- One attribute (`Color`) is a `String Set`.
- The following attributes are document data types:
  - A list of `RelatedItems`. Each element is an `Id` for a related product.
  - A map of `Pictures`. Each element is a short description of a picture, along with a URL for the corresponding image file.
  - A map of `ProductReviews`. Each element represents a rating and a list of reviews corresponding to that rating. Initially, this map is populated with five-star and one-star reviews.

Top-level attributes

An attribute is said to be *top level* if it is not embedded within another attribute. For the `ProductCatalog` item, the top-level attributes are as follows:

- `Id`
- `Title`
- `Description`
- `BicycleType`

- `Brand`
- `Price`
- `Color`
- `ProductCategory`
- `InStock`
- `QuantityOnHand`
- `RelatedItems`
- `Pictures`
- `ProductReviews`
- `Comment`
- `Safety.Warning`

All of these top-level attributes are scalars, except for `Color` (list), `RelatedItems` (list), `Pictures` (map), and `ProductReviews` (map).

Nested attributes

An attribute is said to be *nested* if it is embedded within another attribute. To access a nested attribute, you use *dereference operators*:

- `[n]` — for list elements
- `.` (dot) — for map elements

Accessing list elements

The dereference operator for a list element is **[*N*]**, where *n* is the element number. List elements are zero-based, so [0] represents the first element in the list, [1] represents the second, and so on. Here are some examples:

- `MyList[0]`
- `AnotherList[12]`
- `ThisList[5][11]`

The element `ThisList[5]` is itself a nested list. Therefore, `ThisList[5][11]` refers to the 12th element in that list.

The number within the square brackets must be a non-negative integer. Therefore, the following expressions are not valid:

- `MyList[-1]`
- `MyList[0.4]`

Accessing map elements

The dereference operator for a map element is **.** (a dot). Use a dot as a separator between elements in a map:

- `MyMap.nestedField`
- `MyMap.nestedField.deeplyNestedField`

Document paths

In an expression, you use a *document path* to tell DynamoDB where to find an attribute. For a top-level attribute, the document path is simply the attribute name. For a nested attribute, you construct the document path using dereference operators.

The following are some examples of document paths. (Refer to the item shown in Referring to item attributes when using expressions in DynamoDB.)

- A top-level scalar attribute.

  `Description`
- A top-level list attribute. (This returns the entire list, not just some of the elements.)

  `RelatedItems`
- The third element from the `RelatedItems` list. (Remember that list elements are zero-based.)

  `RelatedItems[2]`
- The front-view picture of the product.

  `Pictures.FrontView`
- All of the five-star reviews.

  `ProductReviews.FiveStar`

- The first of the five-star reviews.

  `ProductReviews.FiveStar[0]`

> **Note**
>
> The maximum depth for a document path is 32. Therefore, the number of dereferences operators in a path cannot exceed this limit.

You can use any attribute name in a document path as long as they meet these requirements:

- The first character is `a-z` or `A-Z` and or `0-9`
- The second character (if present) is `a-z`, `A-Z`

> **Note**
>
> If an attribute name does not meet this requirement, you must define an expression attribute name as a placeholder.

For more information, see Expression attribute names (aliases) in DynamoDB.

Expression attribute names (aliases) in DynamoDB

An *expression attribute name* is an alias (or placeholder) that you use in an Amazon DynamoDB expression as an alternative to an actual attribute name. An expression attribute name must begin with a pound sign (#) and be followed by one or more alphanumeric characters. The underscore (_) character is also allowed.

This section describes several situations in which you must use expression attribute names.

> **Note**
>
> The examples in this section use the AWS Command Line Interface (AWS CLI).

Topics

- Reserved words
- Attribute names containing special characters
- Nested attributes
- Repeatedly referencing attribute names

Reserved words

Sometimes you might need to write an expression containing an attribute name that conflicts with a DynamoDB reserved word. (For a complete list of reserved words, see Reserved words in DynamoDB.)

For example, the following AWS CLI example would fail because `COMMENT` is a reserved word.

```
aws dynamodb get-item \
    --table-name ProductCatalog \
    --key '{"Id":{"N":"123"}}' \
    --projection-expression "Comment"
```

To work around this, you can replace `Comment` with an expression attribute name such as `#c`. The `#` (pound sign) is required and indicates that this is a placeholder for an attribute name. The AWS CLI example would now look like the following.

```
aws dynamodb get-item \
     --table-name ProductCatalog \
     --key '{"Id":{"N":"123"}}' \
     --projection-expression "#c" \
     --expression-attribute-names '{"#c":"Comment"}'
```

> **Note**
>
> If an attribute name begins with a number, contains a space or contains a reserved word, you *must* use an expression attribute name to replace that attribute's name in the expression.

Attribute names containing special characters

In an expression, a dot (".") is interpreted as a separator character in a document path. However, DynamoDB also allows you to use a dot character and other special characters, such as a hyphen

("-") as part of an attribute name. This can be ambiguous in some cases. To illustrate, suppose that you wanted to retrieve the `Safety.Warning` attribute from a `ProductCatalog` item (see Referring to item attributes when using expressions in DynamoDB).

Suppose that you wanted to access `Safety.Warning` using a projection expression.

```
aws dynamodb get-item \
    --table-name ProductCatalog \
    --key '{"Id":{"N":"123"}}' \
    --projection-expression "Safety.Warning"
```

DynamoDB would return an empty result, rather than the expected string ("`Always wear a helmet`"). This is because DynamoDB interprets a dot in an expression as a document path separator. In this case, you must define an expression attribute name (such as `#sw`) as a substitute for `Safety.Warning`. You could then use the following projection expression.

```
aws dynamodb get-item \
    --table-name ProductCatalog \
    --key '{"Id":{"N":"123"}}' \
    --projection-expression "#sw" \
    --expression-attribute-names '{"#sw":"Safety.Warning"}'
```

DynamoDB would then return the correct result.

> **Note**
>
> If an attribute name contains a dot (".") or a hyphen ("-"), you *must* use an expression attribute name to replace that attribute's name in the expression.

Nested attributes

Suppose that you wanted to access the nested attribute `ProductReviews.OneStar`. In an expression attribute name, DynamoDB treats the dot (".") as a character within an attribute's name. To reference the nested attribute, define an expression attribute name for each element in the document path:

- `#pr – ProductReviews`
- `#1star – OneStar`

You could then use `#pr.#1star` for the projection expression.

```
aws dynamodb get-item \
    --table-name ProductCatalog \
    --key '{"Id":{"N":"123"}}' \
    --projection-expression "#pr.#1star"  \
    --expression-attribute-names '{"#pr":"ProductReviews", "#1star":"OneStar"}'
```

DynamoDB would then return the correct result.

Repeatedly referencing attribute names

Expression attribute names are helpful when you need to refer to the same attribute name repeatedly. For example, consider the following expression for retrieving some of the reviews from a `ProductCatalog` item.

```
aws dynamodb get-item \
    --table-name ProductCatalog \
    --key '{"Id":{"N":"123"}}' \
    --projection-expression "ProductReviews.FiveStar, ProductReviews.ThreeStar,
 ProductReviews.OneStar"
```

To make this more concise, you can replace `ProductReviews` with an expression attribute name such as `#pr`. The revised expression would now look like the following.

- `#pr.FiveStar, #pr.ThreeStar, #pr.OneStar`

```
aws dynamodb get-item \
    --table-name ProductCatalog \
    --key '{"Id":{"N":"123"}}' \
    --projection-expression "#pr.FiveStar, #pr.ThreeStar, #pr.OneStar" \
    --expression-attribute-names '{"#pr":"ProductReviews"}'
```

If you define an expression attribute name, you must use it consistently throughout the entire expression. Also, you cannot omit the # symbol.

Using expression attribute values in DynamoDB

Expression attribute values in Amazon DynamoDB act as variables. They're substitutes for the actual values that you want to compare—values that you might not know until runtime. An expression

attribute value must begin with a colon (`:`) and be followed by one or more alphanumeric characters.

For example, suppose that you wanted to return all of the `ProductCatalog` items that are available in `Black` and cost `500` or less. You could use a `Scan` operation with a filter expression, as in this AWS Command Line Interface (AWS CLI) example.

```
aws dynamodb scan \
    --table-name ProductCatalog \
    --filter-expression "contains(Color, :c) and Price <= :p" \
    --expression-attribute-values file://values.json
```

The arguments for `--expression-attribute-values` are stored in the `values.json` file.

```
{
    ":c": { "S": "Black" },
    ":p": { "N": "500" }
}
```

If you define an expression attribute value, you must use it consistently throughout the entire expression. Also, you can't omit the `:` symbol.

Expression attribute values are used with key condition expressions, condition expressions, update expressions, and filter expressions.

Using projection expressions in DynamoDB

To read data from a table, you use operations such as `GetItem`, `Query`, or `Scan`. Amazon DynamoDB returns all the item attributes by default. To get only some, rather than all of the attributes, use a projection expression.

A *projection expression* is a string that identifies the attributes that you want. To retrieve a single attribute, specify its name. For multiple attributes, the names must be comma-separated.

The following are some examples of projection expressions, based on the `ProductCatalog` item from Referring to item attributes when using expressions in DynamoDB:

- A single top-level attribute.

  `Title`

- Three top-level attributes. DynamoDB retrieves the entire `Color` set.

  `Title, Price, Color`

- Four top-level attributes. DynamoDB returns the entire contents of `RelatedItems` and `ProductReviews`.

  `Title, Description, RelatedItems, ProductReviews`

> **Note**
>
> Projection expression has no effect on provisioned throughput consumption. DynamoDB determines capacity units consumed based on item size, instead of the amount of data that is returned to an application.

Reserved words and special characters

DynamoDB has reserved words and special characters. DynamoDB allows you to use these reserved words and special characters for names, but we recommend that you avoid doing so because you have to use aliases for them whenever you use these names in an expression. For a complete list, see Reserved words in DynamoDB.

You'll need to use expression attribute names in place of the actual name if:

- The attribute name is on the list of reserved words in DynamoDB.
- The attribute name does not meet the requirement that the first character is `a-z` or `A-Z` and that the second character (if present) is `a-Z`, `A-Z`, or `0-9`.
- The attribute name contains a **#** (hash) or **:** (colon).

The following AWS CLI example shows how to use a projection expression with a `GetItem` operation. This projection expression retrieves a top-level scalar attribute (`Description`), the first element in a list (`RelatedItems[0]`), and a list nested within a map (`ProductReviews.FiveStar`).

```
aws dynamodb get-item \
    --table-name ProductCatalog \
    --key '"Id": { "N": "123" } \
```

```
    --projection-expression "Description, RelatedItems[0], ProductReviews.FiveStar"
```

The following JSON would be returned for this example.

```
{
    "Item": {
        "Description": {
            "S": "123 description"
        },
        "ProductReviews": {
            "M": {
                "FiveStar": {
                    "L": [
                        {
                            "S": "Excellent! Can't recommend it highly enough! Buy it!"
                        },
                        {
                            "S": "Do yourself a favor and buy this."
                        }
                    ]
                }
            }
        },
        "RelatedItems": {
            "L": [
                {
                    "N": "341"
                }
            ]
        }
    }
}
```

Using update expressions in DynamoDB

The `UpdateItem` operation updates an existing item, or adds a new item to the table if it does not already exist. You must provide the key of the item that you want to update. You must also provide an update expression, indicating the attributes that you want to modify and the values that you want to assign to them.

An *update expression* specifies how `UpdateItem` will modify the attributes of an item—for example, setting a scalar value or removing elements from a list or a map.

The following is a syntax summary for update expressions.

```
update-expression ::=
    [ SET action [, action] ... ]
    [ REMOVE action [, action] ...]
    [ ADD action [, action] ... ]
    [ DELETE action [, action] ...]
```

An update expression consists of one or more clauses. Each clause begins with a `SET`, `REMOVE`, `ADD`, or `DELETE` keyword. You can include any of these clauses in an update expression, in any order. However, each action keyword can appear only once.

Within each clause, there are one or more actions separated by commas. Each action represents a data modification.

The examples in this section are based on the `ProductCatalog` item shown in Using projection expressions in DynamoDB.

The topics below cover some different use cases for the `SET` action.

Topics

- SET — modifying or adding item attributes
- REMOVE — deleting attributes from an item
- ADD — updating numbers and sets
- DELETE — removing elements from a set
- Using multiple update expressions

SET — modifying or adding item attributes

Use the `SET` action in an update expression to add one or more attributes to an item. If any of these attributes already exists, they are overwritten by the new values. If you want to avoid overwriting an existing attribute, you can use `SET` with the `if_not_exists` function. The `if_not_exists` function is specific to the `SET` action and can only be used in an update expression.

When you use `SET` to update a list element, the contents of that element are replaced with the new data that you specify. If the element doesn't already exist, `SET` appends the new element at the end of the list.

If you add multiple elements in a single SET operation, the elements are sorted in order by element number.

You can also use SET to add or subtract from an attribute that is of type `Number`. To perform multiple SET actions, separate them with commas.

In the following syntax summary:

- The *path* element is the document path to the item.
- An *operand* element can be either a document path to an item or a function.

```
set-action ::=
    path = value

value ::=
    operand
    | operand '+' operand
    | operand '-' operand

operand ::=
    path | function

function ::=
    if_not_exists (path, value)
```

If the item does not contain an attribute at the specified path, `if_not_exists` evaluates to `value`. Otherwise, it evaluates to `path`.

The following `PutItem` operation creates a sample item that the examples refer to.

```
aws dynamodb put-item \
    --table-name ProductCatalog \
    --item file://item.json
```

The arguments for `--item` are stored in the `item.json` file. (For simplicity, only a few item attributes are used.)

```
{
    "Id": {"N": "789"},
    "ProductCategory": {"S": "Home Improvement"},
    "Price": {"N": "52"},
```

```
    "InStock": {"BOOL": true},
    "Brand": {"S": "Acme"}
}
```

Topics

- Modifying attributes
- Adding lists and maps
- Adding elements to a list
- Adding nested map attributes
- Incrementing and decrementing numeric attributes
- Appending elements to a list
- Preventing overwrites of an existing attribute

Modifying attributes

Example

Update the `ProductCategory` and `Price` attributes.

```
aws dynamodb update-item \
    --table-name ProductCatalog \
    --key '{"Id":{"N":"789"}}' \
    --update-expression "SET ProductCategory = :c, Price = :p" \
    --expression-attribute-values file://values.json \
    --return-values ALL_NEW
```

The arguments for `--expression-attribute-values` are stored in the `values.json` file.

```
{
    ":c": { "S": "Hardware" },
    ":p": { "N": "60" }
}
```

> **Note**
>
> In the `UpdateItem` operation, `--return-values ALL_NEW` causes DynamoDB to return the item as it appears after the update.

Adding lists and maps

Example

Add a new list and a new map.

```
aws dynamodb update-item \
    --table-name ProductCatalog \
    --key '{"Id":{"N":"789"}}' \
    --update-expression "SET RelatedItems = :ri, ProductReviews = :pr" \
    --expression-attribute-values file://values.json \
    --return-values ALL_NEW
```

The arguments for `--expression-attribute-values` are stored in the `values.json` file.

```
{
    ":ri": {
        "L": [
            { "S": "Hammer" }
        ]
    },
    ":pr": {
        "M": {
            "FiveStar": {
                "L": [
                    { "S": "Best product ever!" }
                ]
            }
        }
    }
}
```

Adding elements to a list

Example

Add a new attribute to the `RelatedItems` list. (Remember that list elements are zero-based, so [0] represents the first element in the list, [1] represents the second, and so on.)

```
aws dynamodb update-item \
    --table-name ProductCatalog \
    --key '{"Id":{"N":"789"}}' \
    --update-expression "SET RelatedItems[1] = :ri" \
```

```
    --expression-attribute-values file://values.json \
    --return-values ALL_NEW
```

The arguments for `--expression-attribute-values` are stored in the `values.json` file.

```
{
    ":ri": { "S": "Nails" }
}
```

> **Note**
>
> When you use SET to update a list element, the contents of that element are replaced with the new data that you specify. If the element doesn't already exist, SET appends the new element at the end of the list.
> If you add multiple elements in a single SET operation, the elements are sorted in order by element number.

Adding nested map attributes

Example

Add some nested map attributes.

```
aws dynamodb update-item \
    --table-name ProductCatalog \
    --key '{"Id":{"N":"789"}}' \
    --update-expression "SET #pr.#5star[1] = :r5, #pr.#3star = :r3" \
    --expression-attribute-names file://names.json \
    --expression-attribute-values file://values.json \
    --return-values ALL_NEW
```

The arguments for `--expression-attribute-names` are stored in the `names.json` file.

```
{
    "#pr": "ProductReviews",
    "#5star": "FiveStar",
    "#3star": "ThreeStar"
}
```

The arguments for `--expression-attribute-values` are stored in the `values.json` file.

```
{
    ":r5": { "S": "Very happy with my purchase" },
    ":r3": {
        "L": [
            { "S": "Just OK - not that great" }
        ]
    }
}
```

Important

You cannot update nested map attributes if the parent map does not exist. If you attempt to update a nested attribute (for example, `ProductReviews.FiveStar`) when the parent map (`ProductReviews`) does not exist, DynamoDB returns a `ValidationException` with the message *"The document path provided in the update expression is invalid for update."*

When creating items that will have nested map attributes updated later, initialize empty maps for the parent attributes. For example:

```
{
    "Id": {"N": "789"},
    "ProductReviews": {"M": {}},
    "Metadata": {"M": {}}
}
```

This allows you to update nested attributes like `ProductReviews.FiveStar` without errors.

Incrementing and decrementing numeric attributes

You can add to or subtract from an existing numeric attribute. To do this, use the + (plus) and - (minus) operators.

Example

Decrease the `Price` of an item.

```
aws dynamodb update-item \
    --table-name ProductCatalog \
```

```
    --key '{"Id":{"N":"789"}}' \
    --update-expression "SET Price = Price - :p" \
    --expression-attribute-values '{":p": {"N":"15"}}' \
    --return-values ALL_NEW
```

To increase the `Price`, you would use the + operator in the update expression.

Appending elements to a list

You can add elements to the end of a list. To do this, use SET with the `list_append` function. (The function name is case sensitive.) The `list_append` function is specific to the SET action and can only be used in an update expression. The syntax is as follows.

- `list_append (`*`list1`*`, `*`list2`*`)`

The function takes two lists as input and appends all elements from *`list2`* to *`list1`*.

Example

In Adding elements to a list, you create the `RelatedItems` list and populate it with two elements: `Hammer` and `Nails`. Now you append two more elements to the end of `RelatedItems`.

```
aws dynamodb update-item \
    --table-name ProductCatalog \
    --key '{"Id":{"N":"789"}}' \
    --update-expression "SET #ri = list_append(#ri, :vals)" \
    --expression-attribute-names '{"#ri": "RelatedItems"}' \
    --expression-attribute-values file://values.json  \
    --return-values ALL_NEW
```

The arguments for `--expression-attribute-values` are stored in the `values.json` file.

```
{
    ":vals": {
        "L": [
            { "S": "Screwdriver" },
            {"S": "Hacksaw" }
        ]
    }
}
```

Finally, you append one more element to the *beginning* of `RelatedItems`. To do this, swap the order of the `list_append` elements. (Remember that `list_append` takes two lists as input and appends the second list to the first.)

```
aws dynamodb update-item \
    --table-name ProductCatalog \
    --key '{"Id":{"N":"789"}}' \
    --update-expression "SET #ri = list_append(:vals, #ri)" \
    --expression-attribute-names '{"#ri": "RelatedItems"}' \
    --expression-attribute-values '{":vals": {"L": [ { "S": "Chisel" }]}}' \
    --return-values ALL_NEW
```

The resulting `RelatedItems` attribute now contains five elements, in the following order: `Chisel`, `Hammer`, `Nails`, `Screwdriver`, `Hacksaw`.

Preventing overwrites of an existing attribute

Example

Set the `Price` of an item, but only if the item does not already have a `Price` attribute. (If `Price` already exists, nothing happens.)

```
aws dynamodb update-item \
    --table-name ProductCatalog \
    --key '{"Id":{"N":"789"}}' \
    --update-expression "SET Price = if_not_exists(Price, :p)" \
    --expression-attribute-values '{":p": {"N": "100"}}' \
    --return-values ALL_NEW
```

REMOVE — deleting attributes from an item

Use the `REMOVE` action in an update expression to remove one or more attributes from an item in Amazon DynamoDB. To perform multiple `REMOVE` actions, separate them with commas.

The following is a syntax summary for `REMOVE` in an update expression. The only operand is the document path for the attribute that you want to remove.

```
remove-action ::=
    path
```

Example

Remove some attributes from an item. (If the attributes don't exist, nothing happens.)

```
aws dynamodb update-item \
    --table-name ProductCatalog \
    --key '{"Id":{"N":"789"}}' \
    --update-expression "REMOVE Brand, InStock, QuantityOnHand" \
    --return-values ALL_NEW
```

Removing elements from a list

You can use `REMOVE` to delete individual elements from a list.

Example

In Appending elements to a list, you modify a list attribute (`RelatedItems`) so that it contained five elements:

- `[0]—Chisel`
- `[1]—Hammer`
- `[2]—Nails`
- `[3]—Screwdriver`
- `[4]—Hacksaw`

The following AWS Command Line Interface (AWS CLI) example deletes `Hammer` and `Nails` from the list.

```
aws dynamodb update-item \
    --table-name ProductCatalog \
    --key '{"Id":{"N":"789"}}' \
    --update-expression "REMOVE RelatedItems[1], RelatedItems[2]" \
    --return-values ALL_NEW
```

After `Hammer` and `Nails` are removed, the remaining elements are shifted. The list now contains the following:

- `[0]—Chisel`

- `[1]—Screwdriver`
- `[2]—Hacksaw`

ADD — updating numbers and sets

> **Note**
>
> In general, we recommend using SET rather than ADD to ensure idempotent operations.

Use the ADD action in an update expression to add a new attribute and its values to an item.

If the attribute already exists, the behavior of ADD depends on the attribute's data type:

- If the attribute is a number, and the value you are adding is also a number, the value is mathematically added to the existing attribute. (If the value is a negative number, it is subtracted from the existing attribute.)
- If the attribute is a set, and the value you are adding is also a set, the value is appended to the existing set.

> **Note**
>
> The ADD action supports only number and set data types.

To perform multiple ADD actions, separate them with commas.

In the following syntax summary:

- The *`path`* element is the document path to an attribute. The attribute must be either a `Number` or a set data type.
- The *`value`* element is a number that you want to add to the attribute (for `Number` data types), or a set to append to the attribute (for set types).

```
add-action ::=
    path value
```

The topics below cover some different use cases for the ADD action.

Topics

- Adding a number
- Adding elements to a set

Adding a number

Assume that the QuantityOnHand attribute does not exist. The following AWS CLI example sets QuantityOnHand to 5.

```
aws dynamodb update-item \
    --table-name ProductCatalog \
    --key '{"Id":{"N":"789"}}' \
    --update-expression "ADD QuantityOnHand :q" \
    --expression-attribute-values '{":q": {"N": "5"}}' \
    --return-values ALL_NEW
```

Now that QuantityOnHand exists, you can rerun the example to increment QuantityOnHand by 5 each time.

Adding elements to a set

Assume that the Color attribute does not exist. The following AWS CLI example sets Color to a string set with two elements.

```
aws dynamodb update-item \
    --table-name ProductCatalog \
    --key '{"Id":{"N":"789"}}' \
    --update-expression "ADD Color :c" \
    --expression-attribute-values '{":c": {"SS":["Orange", "Purple"]}}' \
    --return-values ALL_NEW
```

Now that Color exists, you can add more elements to it.

```
aws dynamodb update-item \
    --table-name ProductCatalog \
    --key '{"Id":{"N":"789"}}' \
    --update-expression "ADD Color :c" \
```

```
    --expression-attribute-values '{":c": {"SS":["Yellow", "Green", "Blue"]}}' \
    --return-values ALL_NEW
```

DELETE — removing elements from a set

> **Important**
>
> The DELETE action supports only Set data types.

Use the DELETE action in an update expression to remove one or more elements from a set. To perform multiple DELETE actions, separate them with commas.

In the following syntax summary:

- The *path* element is the document path to an attribute. The attribute must be a set data type.
- The *subset* is one or more elements that you want to delete from *path*. You must specify *subset* as a set type.

```
delete-action ::=
    path subset
```

Example

In Adding elements to a set, you create the Color string set. This example removes some of the elements from that set.

```
aws dynamodb update-item \
    --table-name ProductCatalog \
    --key '{"Id":{"N":"789"}}' \
    --update-expression "DELETE Color :p" \
    --expression-attribute-values '{":p": {"SS": ["Yellow", "Purple"]}}' \
    --return-values ALL_NEW
```

Using multiple update expressions

You can use multiple actions in a single update expression. All attribute references are resolved against the item's state before any of the actions are applied.

Example

Given an item `{"id": "1", "a": 1, "b": 2, "c": 3}`, the following expression removes a and shifts the values of b and c:

```
aws dynamodb update-item \
    --table-name test \
    --key '{"id":{"S":"1"}}' \
    --update-expression "REMOVE a SET b = a, c = b" \
    --return-values ALL_NEW
```

The result is `{"id": "1", "b": 1, "c": 2}`. Even though a is removed and b is reassigned in the same expression, both references resolve to their original values.

Example

If you want to modify an attribute's value and completely remove another attribute, you could use a SET and a REMOVE action in a single statement. This operation would reduce the `Price` value to 15 while also removing the `InStock` attribute from the item.

```
aws dynamodb update-item \
    --table-name ProductCatalog \
    --key '{"Id":{"N":"789"}}' \
    --update-expression "SET Price = Price - :p REMOVE InStock" \
    --expression-attribute-values '{":p": {"N":"15"}}' \
    --return-values ALL_NEW
```

Example

If you want to add to a list while also changing another attribute's value, you could use two SET actions in a single statement. This operation would add "Nails" to the `RelatedItems` list attribute and also set the `Price` value to 21.

```
aws dynamodb update-item \
    --table-name ProductCatalog \
    --key '{"Id":{"N":"789"}}' \
    --update-expression "SET RelatedItems[1] = :newValue, Price = :newPrice" \
    --expression-attribute-values '{":newValue": {"S":"Nails"}, ":newPrice":
 {"N":"21"}}'  \
    --return-values ALL_NEW
```

Condition and filter expressions, operators, and functions in DynamoDB

To manipulate data in an DynamoDB table, you use the `PutItem`, `UpdateItem`, and `DeleteItem` operations. For these data manipulation operations, you can specify a condition expression to determine which items should be modified. If the condition expression evaluates to true, the operation succeeds. Otherwise, the operation fails.

This section covers the built-in functions and keywords for writing filter expressions and condition expressions in Amazon DynamoDB. For more detailed information on functions and programming with DynamoDB, see Programming with DynamoDB and the AWS SDKs and the DynamoDB API Reference.

Topics

- Syntax for filter and condition expressions
- Making comparisons
- Functions
- Logical evaluations
- Parentheses
- Precedence in conditions

Syntax for filter and condition expressions

In the following syntax summary, an *operand* can be the following:

- A top-level attribute name, such as `Id`, `Title`, `Description`, or `ProductCategory`
- A document path that references a nested attribute

```
condition-expression ::=
      operand comparator operand
    | operand BETWEEN operand AND operand
    | operand IN ( operand (',' operand (, ...) ))
    | function
    | condition AND condition
    | condition OR condition
    | NOT condition
    | ( condition )

comparator ::=
```

```
    =
    | <>
    | <
    | <=
    | >
    | >=

function ::=
    attribute_exists (path)
    | attribute_not_exists (path)
    | attribute_type (path, type)
    | begins_with (path, substr)
    | contains (path, operand)
    | size (path)
```

Making comparisons

Use these comparators to compare an operand against a single value:

- *a* = *b* – True if *a* is equal to *b*.
- *a* <> *b* – True if *a* is not equal to *b*.
- *a* < *b* – True if *a* is less than *b*.
- *a* <= *b* – True if *a* is less than or equal to *b*.
- *a* > *b* – True if *a* is greater than *b*.
- *a* >= *b* – True if *a* is greater than or equal to *b*.

Use the `BETWEEN` and `IN` keywords to compare an operand against a range of values or an enumerated list of values:

- *a* `BETWEEN` *b* `AND` *c* – True if *a* is greater than or equal to *b*, and less than or equal to *c*.
- *a* `IN (`*b*`, `*c*`, `*d*`)` – True if *a* is equal to any value in the list—for example, any of *b*, *c*, or *d*. The list can contain up to 100 values, separated by commas.

Functions

Use the following functions to determine whether an attribute exists in an item, or to evaluate the value of an attribute. These function names are case sensitive. For a nested attribute, you must provide its full document path.

Function	Description
`attribute_exists ( path)`	True if the item contains the attribute specified by `path`. Example: Check whether an item in the `Product` table has a side view picture. • `attribute_exists (#Pictures.#SideView)`
`attribute_not_exists ( path)`	True if the attribute specified by `path` does not exist in the item. Example: Check whether an item has a `Manufacturer` attribute. • `attribute_not_exists (Manufacturer)`
`attribute_type ( path, type)`	True if the attribute at the specified path is of a particular data type. The `type` parameter must be one of the following: • S – String • SS – String set • N – Number • NS – Number set • B – Binary • BS – Binary set • BOOL – Boolean

Function	Description
	• `NULL` – Null • `L` – List • `M` – Map You must use an expression attribute value for the `type` parameter. Example: Check whether the `Quantity0nHand` attribute is of type List. In this example, `:v_sub` is a placeholder for the string `L`. • `attribute_type (ProductReviews.FiveStar, :v_sub)` You must use an expression attribute value for the `type` parameter.
`begins_with ( path, substr)`	True if the attribute specified by `path` begins with a particular substring. Example: Check whether the first few characters of the front view picture URL are `http://`. • `begins_with (Pictures.FrontView, :v_sub)` The expression attribute value `:v_sub` is a placeholder for `http://`.

Function	Description
`contains (path, operand)`	True if the attribute specified by `path` is one of the following: • A `String` that contains a particular substring. • A `Set` that contains a particular element within the set. • A `List` that contains a particular element within the list. If the attribute specified by `path` is a `String`, the `operand` must be a `String`. If the attribute specified by `path` is a `Set`, the `operand` must be the set's element type. The path and the operand must be distinct. That is, `contains (a, a)` returns an error. Example: Check whether the `Brand` attribute contains the substring `Company`. • `contains (Brand, :v_sub)` The expression attribute value `:v_sub` is a placeholder for `Company`. Example: Check whether the product is available in red. • `contains (Color, :v_sub)`

Function	Description
	The expression attribute value `:v_sub` is a placeholder for `Red`.

Function	Description
`size (path)`	Returns a number that represents an attribute 's size. The following are valid data types for use with `size`. If the attribute is of type `String`, `size` returns the length of the string. Example: Check whether the string `Brand` is less than or equal to 20 characters. The expression attribute value `:v_sub` is a placeholder for `20`. • `size (Brand) <= :v_sub` If the attribute is of type `Binary`, `size` returns the number of bytes in the attribute value. Example: Suppose that the `ProductCatalog` item has a binary attribute named `VideoClip` that contains a short video of the product in use. The following expression checks whether `VideoClip` exceeds 64,000 bytes. The expression attribute value `:v_sub` is a placeholder for `64000`. • `size(VideoClip) > :v_sub` If the attribute is a `Set` data type, `size` returns the number of elements in the set.

Function	Description
	Example: Check whether the product is available in more than one color. The expression attribute value `:v_sub` is a placeholder for `1`. • `size (Color) < :v_sub` If the attribute is of type `List` or `Map`, `size` returns the number of child elements. Example: Check whether the number of `OneStar` reviews has exceeded a certain threshold. The expression attribute value `:v_sub` is a placeholder for `3`. • `size(ProductReviews.OneStar) > :v_sub`

Logical evaluations

Use the `AND`, `OR`, and `NOT` keywords to perform logical evaluations. In the following list, *a* and *b* represent conditions to be evaluated.

- *a* `AND` *b* – True if *a* and *b* are both true.
- *a* `OR` *b* – True if either *a* or *b* (or both) are true.
- `NOT` *a* – True if *a* is false. False if *a* is true.

The following is a code example of AND in an operation.

```
dynamodb-local (*)> select * from exprtest where a > 3 and a < 5;
```

Parentheses

Use parentheses to change the precedence of a logical evaluation. For example, suppose that conditions *a* and *b* are true, and that condition *c* is false. The following expression evaluates to true:

- `a OR b AND c`

However, if you enclose a condition in parentheses, it is evaluated first. For example, the following evaluates to false:

- `(a OR b) AND c`

> **Note**
>
> You can nest parentheses in an expression. The innermost ones are evaluated first.

The following is a code example with parentheses in a logical evaluation.

```
dynamodb-local (*)> select * from exprtest where attribute_type(b, string)
or ( a = 5 and c = “coffee”);
```

Precedence in conditions

DynamoDB evaluates conditions from left to right using the following precedence rules:

- `= <> < <= > >=`
- `IN`
- `BETWEEN`
- `attribute_exists attribute_not_exists begins_with contains`
- Parentheses
- `NOT`
- `AND`
- `OR`

DynamoDB condition expression CLI example

The following are some AWS Command Line Interface (AWS CLI) examples of using condition expressions. These examples are based on the `ProductCatalog` table, which was introduced in Referring to item attributes when using expressions in DynamoDB. The partition key for this table is `Id`; there is no sort key. The following `PutItem` operation creates a sample `ProductCatalog` item that the examples refer to.

```
aws dynamodb put-item \
    --table-name ProductCatalog \
    --item file://item.json
```

The arguments for `--item` are stored in the `item.json` file. (For simplicity, only a few item attributes are used.)

```
{
    "Id": {"N": "456" },
    "ProductCategory": {"S": "Sporting Goods" },
    "Price": {"N": "650" }
}
```

Topics

- Conditional put
- Conditional deletes
- Conditional updates
- Conditional expression examples

Conditional put

The `PutItem` operation overwrites an item with the same primary key (if it exists). If you want to avoid this, use a condition expression. This allows the write to proceed only if the item in question does not already have the same primary key.

The following example uses `attribute_not_exists()` to check whether the primary key exists in the table before attempting the write operation.

> **Note**
>
> If your primary key consists of both a partition key(pk) and a sort key(sk), the parameter will check whether `attribute_not_exists(pk)` AND `attribute_not_exists(sk)` evaluate to true or false as an entire statement before attempting the write operation.

```
aws dynamodb put-item \
    --table-name ProductCatalog \
    --item file://item.json \
    --condition-expression "attribute_not_exists(Id)"
```

If the condition expression evaluates to false, DynamoDB returns the following error message: The conditional request failed.

> **Note**
>
> For more information about `attribute_not_exists` and other functions, see Condition and filter expressions, operators, and functions in DynamoDB.

Conditional deletes

To perform a conditional delete, you use a `DeleteItem` operation with a condition expression. The condition expression must evaluate to true in order for the operation to succeed; otherwise, the operation fails.

Consider the item defined above.

Suppose that you wanted to delete the item, but only under the following conditions:

- The `ProductCategory` is either "Sporting Goods" or "Gardening Supplies."
- The `Price` is between 500 and 600.

The following example tries to delete the item.

```
aws dynamodb delete-item \
    --table-name ProductCatalog \
```

```
    --key '{"Id":{"N":"456"}}' \
    --condition-expression "(ProductCategory IN (:cat1, :cat2)) and (Price between :lo
 and :hi)" \
    --expression-attribute-values file://values.json
```

The arguments for `--expression-attribute-values` are stored in the `values.json` file.

```
{
    ":cat1": {"S": "Sporting Goods"},
    ":cat2": {"S": "Gardening Supplies"},
    ":lo": {"N": "500"},
    ":hi": {"N": "600"}
}
```

> **Note**
>
> In the condition expression, the `:` (colon character) indicates an *expression attribute value*—a placeholder for an actual value. For more information, see Using expression attribute values in DynamoDB.
> For more information about `IN`, `AND`, and other keywords, see Condition and filter expressions, operators, and functions in DynamoDB.

In this example, the `ProductCategory` comparison evaluates to true, but the `Price` comparison evaluates to false. This causes the condition expression to evaluate to false and the `DeleteItem` operation to fail.

Conditional updates

To perform a conditional update, you use an `UpdateItem` operation with a condition expression. The condition expression must evaluate to true in order for the operation to succeed; otherwise, the operation fails.

> **Note**
>
> `UpdateItem` also supports *update expressions*, where you specify the modifications you want to make to an item. For more information, see Using update expressions in DynamoDB.

Suppose that you started with the item defined above.

The following example performs an `UpdateItem` operation. It tries to reduce the `Price` of a product by 75—but the condition expression prevents the update if the current `Price` is less than or equal to 500.

```
aws dynamodb update-item \
    --table-name ProductCatalog \
    --key '{"Id": {"N": "456"}}' \
    --update-expression "SET Price = Price - :discount" \
    --condition-expression "Price > :limit" \
    --expression-attribute-values file://values.json
```

The arguments for `--expression-attribute-values` are stored in the `values.json` file.

```
{
    ":discount": { "N": "75"},
    ":limit": {"N": "500"}
}
```

If the starting `Price` is 650, the `UpdateItem` operation reduces the `Price` to 575. If you run the `UpdateItem` operation again, the `Price` is reduced to 500. If you run it a third time, the condition expression evaluates to false, and the update fails.

> **Note**
>
> In the condition expression, the `:` (colon character) indicates an *expression attribute value*—a placeholder for an actual value. For more information, see Using expression attribute values in DynamoDB.
> For more information about ">" and other operators, see Condition and filter expressions, operators, and functions in DynamoDB.

Conditional expression examples

For more information about the functions used in the following examples, see Condition and filter expressions, operators, and functions in DynamoDB. If you want to know more about how to specify different attribute types in an expression, see Referring to item attributes when using expressions in DynamoDB.

Checking for attributes in an item

You can check for the existence (or nonexistence) of any attribute. If the condition expression evaluates to true, the operation succeeds; otherwise, it fails.

The following example uses `attribute_not_exists` to delete a product only if it does not have a `Price` attribute.

```
aws dynamodb delete-item \
    --table-name ProductCatalog \
    --key '{"Id": {"N": "456"}}' \
    --condition-expression "attribute_not_exists(Price)"
```

DynamoDB also provides an `attribute_exists` function. The following example deletes a product only if it has received poor reviews.

```
aws dynamodb delete-item \
    --table-name ProductCatalog \
    --key '{"Id": {"N": "456"}}' \
    --condition-expression "attribute_exists(ProductReviews.OneStar)"
```

Checking for attribute type

You can check the data type of an attribute value by using the `attribute_type` function. If the condition expression evaluates to true, the operation succeeds; otherwise, it fails.

The following example uses `attribute_type` to delete a product only if it has a `Color` attribute of type String Set.

```
aws dynamodb delete-item \
    --table-name ProductCatalog \
    --key '{"Id": {"N": "456"}}' \
    --condition-expression "attribute_type(Color, :v_sub)" \
    --expression-attribute-values file://expression-attribute-values.json
```

The arguments for `--expression-attribute-values` are stored in the expression-attribute-values.json file.

```
{
    ":v_sub":{"S":"SS"}
}
```

Checking string starting value

You can check if a String attribute value begins with a particular substring by using the `begins_with` function. If the condition expression evaluates to true, the operation succeeds; otherwise, it fails.

The following example uses `begins_with` to delete a product only if the `FrontView` element of the `Pictures` map starts with a specific value.

```
aws dynamodb delete-item \
    --table-name ProductCatalog \
    --key '{"Id": {"N": "456"}}' \
    --condition-expression "begins_with(Pictures.FrontView, :v_sub)" \
    --expression-attribute-values file://expression-attribute-values.json
```

The arguments for `--expression-attribute-values` are stored in the expression-attribute-values.json file.

```
{
    ":v_sub":{"S":"http://"}
}
```

Checking for an element in a set

You can check for an element in a set or look for a substring within a string by using the `contains` function. If the condition expression evaluates to true, the operation succeeds; otherwise, it fails.

The following example uses `contains` to delete a product only if the `Color` String Set has an element with a specific value.

```
aws dynamodb delete-item \
    --table-name ProductCatalog \
    --key '{"Id": {"N": "456"}}' \
    --condition-expression "contains(Color, :v_sub)" \
    --expression-attribute-values file://expression-attribute-values.json
```

The arguments for `--expression-attribute-values` are stored in the expression-attribute-values.json file.

```
{
```

```
    ":v_sub":{"S":"Red"}
}
```

Checking the size of an attribute value

You can check for the size of an attribute value by using the `size` function. If the condition expression evaluates to true, the operation succeeds; otherwise, it fails.

The following example uses `size` to delete a product only if the size of the `VideoClip` Binary attribute is greater than `64000` bytes.

```
aws dynamodb delete-item \
    --table-name ProductCatalog \
    --key '{"Id": {"N": "456"}}' \
    --condition-expression "size(VideoClip) > :v_sub" \
    --expression-attribute-values file://expression-attribute-values.json
```

The arguments for `--expression-attribute-values` are stored in the expression-attribute-values.json file.

```
{
    ":v_sub":{"N":"64000"}
}
```

Using time to live (TTL) in DynamoDB

Time To Live (TTL) for DynamoDB is a cost-effective method for deleting items that are no longer relevant. TTL allows you to define a per-item expiration timestamp that indicates when an item is no longer needed. DynamoDB automatically deletes expired items within a few days of their expiration time, without consuming write throughput.

To use TTL, first enable it on a table and then define a specific attribute to store the TTL expiration timestamp. The timestamp must be stored as a Number data type in Unix epoch time format at the seconds granularity. Items with a TTL attribute that is not a Number type are ignored by the TTL process. Each time an item is created or updated, you can compute the expiration time and save it in the TTL attribute.

Items with valid, expired TTL attributes may be deleted by the system at any time, typically within a few days of their expiration. You can still update the expired items that are pending deletion, including changing or removing their TTL attributes. While updating an expired item,

we recommended that you use a condition expression to make sure the item has not been subsequently deleted. Use filter expressions to remove expired items from Scan and Query results.

Deleted items work similarly to those deleted through typical delete operations. Once deleted, items go into DynamoDB Streams as service deletions instead of user deletes, and are removed from local secondary indexes and global secondary indexes just like other delete operations.

If you are using Global Tables version 2019.11.21 (Current) of global tables and you also use the TTL feature, DynamoDB replicates TTL deletes to all replica tables. The initial TTL delete does not consume Write Capacity Units (WCU) in the region in which the TTL expiry occurs. However, the replicated TTL delete to the replica table(s) consumes a replicated Write Capacity Unit when using provisioned capacity, or Replicated Write Unit when using on-demand capacity mode, in each of the replica regions and applicable charges will apply.

For more information about TTL, see these topics:

Topics

- Enable time to live (TTL) in DynamoDB
- Computing time to live (TTL) in DynamoDB
- Working with expired items and time to live (TTL)

Enable time to live (TTL) in DynamoDB

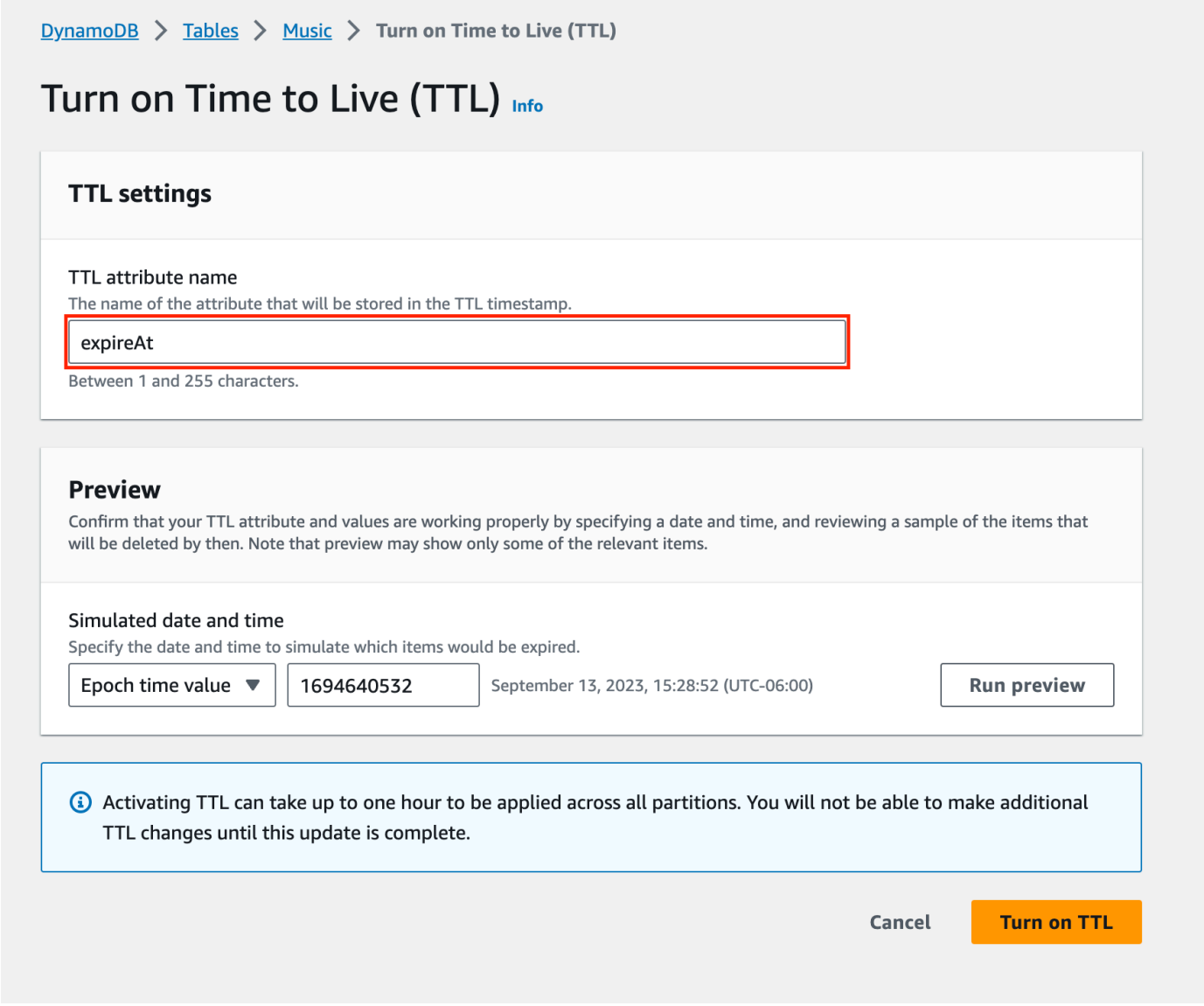
> **Note**
>
> To assist in debugging and verification of proper operation of the TTL feature, the values provided for the item TTL are logged in plaintext in DynamoDB diagnostic logs.

You can enable TTL in the Amazon DynamoDB Console, the AWS Command Line Interface (AWS CLI), or using the Amazon DynamoDB API Reference with any of the supposed AWS SDKs. It takes approximately one hour to enable TTL across all partitions.

Enable DynamoDB TTL using the AWS console

1. Sign in to the AWS Management Console and open the DynamoDB console at https://console.aws.amazon.com/dynamodb/.
2. Choose **Tables**, and then choose the table that you want to modify.

3. In the **Additional settings** tab, in the **Time to Live (TTL)** section, choose **Turn on** to enable TTL.
4. When enabling TTL on a table, DynamoDB requires you to identify a specific attribute name that the service will look for when determining if an item is eligible for expiration. The TTL attribute name, shown below, is case sensitive and must match the attribute defined in your read and write operations. A mismatch will cause expired items to go undeleted. Renaming the TTL attribute requires you to disable TTL and then re-enable it with the new attribute going forward. TTL will continue to process deletions for approximately 30 minutes once it is disabled. TTL must be reconfigured on restored tables.



5. (Optional) You can perform a test by simulating the date and time of the expiration and matching a few items. This provides you with a sample list of items and confirms that there are items containing the TTL attribute name provided along with the expiration time.

After TTL is enabled, the TTL attribute is marked **TTL** when you view items on the DynamoDB console. You can view the date and time that an item expires by hovering your pointer over the attribute.

Enable DynamoDB TTL using the API

Python

You can enable TTL with code, using the UpdateTimeToLive operation.

```
import boto3


def enable_ttl(table_name, ttl_attribute_name):
    """
    Enables TTL on DynamoDB table for a given attribute name
        on success, returns a status code of 200
        on error, throws an exception

    :param table_name: Name of the DynamoDB table
    :param ttl_attribute_name: The name of the TTL attribute being provided to the
 table.
    """
    try:
        dynamodb = boto3.client('dynamodb')

        # Enable TTL on an existing DynamoDB table
        response = dynamodb.update_time_to_live(
            TableName=table_name,
            TimeToLiveSpecification={
                'Enabled': True,
                'AttributeName': ttl_attribute_name
            }
        )

        # In the returned response, check for a successful status code.
        if response['ResponseMetadata']['HTTPStatusCode'] == 200:
            print("TTL has been enabled successfully.")
        else:
            print(f"Failed to enable TTL, status code {response['ResponseMetadata']
['HTTPStatusCode']}")
    except Exception as ex:
        print("Couldn't enable TTL in table %s. Here's why: %s" % (table_name, ex))
```

```
        raise


# your values
enable_ttl('your-table-name', 'expirationDate')
```

You can confirm TTL is enabled by using the DescribeTimeToLive operation, which describes the TTL status on a table. The `TimeToLive` status is either `ENABLED` or `DISABLED`.

```
# create a DynamoDB client
dynamodb = boto3.client('dynamodb')

# set the table name
table_name = 'YourTable'

# describe TTL
response = dynamodb.describe_time_to_live(TableName=table_name)
```

JavaScript

You can enable TTL with code, using the UpdateTimeToLiveCommand operation.

```
import { DynamoDBClient, UpdateTimeToLiveCommand } from "@aws-sdk/client-dynamodb";

const enableTTL = async (tableName, ttlAttribute) => {

    const client = new DynamoDBClient({});

    const params = {
        TableName: tableName,
        TimeToLiveSpecification: {
            Enabled: true,
            AttributeName: ttlAttribute
        }
    };

    try {
        const response = await client.send(new UpdateTimeToLiveCommand(params));
        if (response.$metadata.httpStatusCode === 200) {
            console.log(`TTL enabled successfully for table ${tableName}, using
 attribute name ${ttlAttribute}.`);
        } else {
```

```
            console.log(`Failed to enable TTL for table ${tableName}, response
 object: ${response}`);
        }
        return response;
    } catch (e) {
        console.error(`Error enabling TTL: ${e}`);
        throw e;
    }
};

// call with your own values
enableTTL('ExampleTable', 'exampleTtlAttribute');
```

Enable Time to Live using the AWS CLI

1. Enable TTL on the `TTLExample` table.

```
aws dynamodb update-time-to-live --table-name TTLExample --time-to-live-
specification "Enabled=true, AttributeName=ttl"
```

2. Describe TTL on the `TTLExample` table.

```
aws dynamodb describe-time-to-live --table-name TTLExample
{
    "TimeToLiveDescription": {
        "AttributeName": "ttl",
        "TimeToLiveStatus": "ENABLED"
    }
}
```

3. Add an item to the `TTLExample` table with the Time to Live attribute set using the BASH shell and the AWS CLI.

```
EXP=`date -d '+5 days' +%s`
aws dynamodb put-item --table-name "TTLExample" --item '{"id": {"N": "1"}, "ttl":
 {"N": "'$EXP'"}}'
```

This example starts with the current date and adds 5 days to it to create an expiration time. Then, it converts the expiration time to epoch time format to finally add an item to the "`TTLExample`" table.

> **Note**
>
> One way to set expiration values for Time to Live is to calculate the number of seconds to add to the expiration time. For example, 5 days is 432,000 seconds. However, it is often preferable to start with a date and work from there.

It is fairly simple to get the current time in epoch time format, as in the following examples.

- Linux Terminal: `date +%s`
- Python: `import time; int(time.time())`
- Java: `System.currentTimeMillis() / 1000L`
- JavaScript: `Math.floor(Date.now() / 1000)`

Enable DynamoDB TTL using CloudFormation

```
AWSTemplateFormatVersion: "2010-09-09"
Resources:
  TTLExampleTable:
    Type: AWS::DynamoDB::Table
    Description: "A DynamoDB table with TTL Specification enabled"
    Properties:
      AttributeDefinitions:
        - AttributeName: "Album"
          AttributeType: "S"
        - AttributeName: "Artist"
          AttributeType: "S"
      KeySchema:
        - AttributeName: "Album"
          KeyType: "HASH"
        - AttributeName: "Artist"
          KeyType: "RANGE"
      ProvisionedThroughput:
        ReadCapacityUnits: "5"
        WriteCapacityUnits: "5"
      TimeToLiveSpecification:
        AttributeName: "TTLExampleAttribute"
        Enabled: true
```

Additional details on using TTL within your CloudFormation templates can be found here.

Computing time to live (TTL) in DynamoDB

A common way to implement TTL is to set an expiration time for items based on when they were created or last updated. This can be done by adding time to the `createdAt` and `updatedAt` timestamps. For example, the TTL for newly created items can be set to `createdAt` + 90 days. When the item is updated the TTL can be recalculated to `updatedAt` + 90 days.

The computed expiration time must be in epoch format, in seconds. To be considered for expiry and deletion, the TTL can't be more than five years in the past. If you use any other format, the TTL processes ignore the item. If you set the expiration time to sometime in the future when you want the item to expire, the item expires after that time. For example, say that you set the expiration time to 1724241326 (which is Monday, August 21, 2024 11:55:26 (UTC)). The item expires after the specified time. There is no minimum TTL duration. You can set the expiration time to any future time, such as 5 minutes from the current time. However, DynamoDB typically deletes expired items within 48 hours after their expiration time, not immediately when the item expires.

Topics

- Create an item and set the Time to Live
- Update an item and refresh the Time to Live

Create an item and set the Time to Live

The following example demonstrates how to calculate the expiration time when creating a new item, using `expireAt` as the TTL attribute name. An assignment statement obtains the current time as a variable. In the example, the expiration time is calculated as 90 days from the current time. The time is then converted to epoch format and saved as an integer data type in the TTL attribute.

The following code examples show how to create an item with TTL.

Java

SDK for Java 2.x

```
package com.amazon.samplelib.ttl;

import com.amazon.samplelib.CodeSampleUtils;
import software.amazon.awssdk.regions.Region;
```

```
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.PutItemRequest;
import software.amazon.awssdk.services.dynamodb.model.PutItemResponse;
import software.amazon.awssdk.services.dynamodb.model.ResourceNotFoundException;

import java.util.HashMap;
import java.util.Map;
import java.util.Optional;

/**
 * Creates an item in a DynamoDB table with TTL attributes.
 * This class demonstrates how to add TTL expiration timestamps to DynamoDB
 items.
 */
public class CreateTTL {

    private static final String USAGE =
        """
            Usage:
                <tableName> <primaryKey> <sortKey> <region>
            Where:
                tableName - The Amazon DynamoDB table being queried.
                primaryKey - The name of the primary key. Also known as the hash
 or partition key.
                sortKey - The name of the sort key. Also known as the range
 attribute.
                region (optional) - The AWS region that the Amazon DynamoDB table
 is located in. (Default: us-east-1)
            """;
    private static final int DAYS_TO_EXPIRE = 90;
    private static final int SECONDS_PER_DAY = 24 * 60 * 60;
    private static final String PRIMARY_KEY_ATTR = "primaryKey";
    private static final String SORT_KEY_ATTR = "sortKey";
    private static final String CREATION_DATE_ATTR = "creationDate";
    private static final String EXPIRE_AT_ATTR = "expireAt";
    private static final String SUCCESS_MESSAGE = "%s PutItem operation with TTL
 successful.";
    private static final String TABLE_NOT_FOUND_ERROR = "Error: The Amazon
 DynamoDB table \"%s\" can't be found.";

    private final DynamoDbClient dynamoDbClient;
```

```
    /**
     * Constructs a CreateTTL instance with the specified DynamoDB client.
     *
     * @param dynamoDbClient The DynamoDB client to use
     */
    public CreateTTL(final DynamoDbClient dynamoDbClient) {
        this.dynamoDbClient = dynamoDbClient;
    }

    /**
     * Constructs a CreateTTL with a default DynamoDB client.
     */
    public CreateTTL() {
        this.dynamoDbClient = null;
    }

    /**
     * Main method to demonstrate creating an item with TTL.
     *
     * @param args Command line arguments
     */
    public static void main(final String[] args) {
        try {
            int result = new CreateTTL().processArgs(args);
            System.exit(result);
        } catch (Exception e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }

    /**
     * Process command line arguments and create an item with TTL.
     *
     * @param args Command line arguments
     * @return 0 if successful, non-zero otherwise
     * @throws ResourceNotFoundException If the table doesn't exist
     * @throws DynamoDbException If an error occurs during the operation
     * @throws IllegalArgumentException If arguments are invalid
     */
    public int processArgs(final String[] args) {
        // Argument validation (remove or replace this line when reusing this
code)
        CodeSampleUtils.validateArgs(args, new int[] {3, 4}, USAGE);
```

```
        final String tableName = args[0];
        final String primaryKey = args[1];
        final String sortKey = args[2];
        final Region region = Optional.ofNullable(args.length > 3 ? args[3] :
null)
            .map(Region::of)
            .orElse(Region.US_EAST_1);

        try (DynamoDbClient ddb = dynamoDbClient != null
            ? dynamoDbClient
            : DynamoDbClient.builder().region(region).build()) {
            final CreateTTL createTTL = new CreateTTL(ddb);
            createTTL.createItemWithTTL(tableName, primaryKey, sortKey);
            return 0;
        } catch (Exception e) {
            throw e;
        }
    }

    /**
     * Creates an item in the specified table with TTL attributes.
     *
     * @param tableName The name of the table
     * @param primaryKeyValue The value for the primary key
     * @param sortKeyValue The value for the sort key
     * @return The response from the PutItem operation
     * @throws ResourceNotFoundException If the table doesn't exist
     * @throws DynamoDbException If an error occurs during the operation
     */
    public PutItemResponse createItemWithTTL(
        final String tableName, final String primaryKeyValue, final String
sortKeyValue) {
        // Get current time in epoch second format
        final long createDate = System.currentTimeMillis() / 1000;

        // Calculate expiration time 90 days from now in epoch second format
        final long expireDate = createDate + (DAYS_TO_EXPIRE * SECONDS_PER_DAY);

        final Map<String, AttributeValue> itemMap = new HashMap<>();
        itemMap.put(
            PRIMARY_KEY_ATTR,
AttributeValue.builder().s(primaryKeyValue).build());
```

```
        itemMap.put(SORT_KEY_ATTR,
 AttributeValue.builder().s(sortKeyValue).build());
        itemMap.put(
            CREATION_DATE_ATTR,
            AttributeValue.builder().n(String.valueOf(createDate)).build());
        itemMap.put(
            EXPIRE_AT_ATTR,
            AttributeValue.builder().n(String.valueOf(expireDate)).build());

        final PutItemRequest request =
            PutItemRequest.builder().tableName(tableName).item(itemMap).build();

        try {
            final PutItemResponse response = dynamoDbClient.putItem(request);
            System.out.println(String.format(SUCCESS_MESSAGE, tableName));
            return response;
        } catch (ResourceNotFoundException e) {
            System.err.format(TABLE_NOT_FOUND_ERROR, tableName);
            throw e;
        } catch (DynamoDbException e) {
            System.err.println(e.getMessage());
            throw e;
        }
    }
}
```

- For API details, see PutItem in *AWS SDK for Java 2.x API Reference*.

JavaScript

SDK for JavaScript (v3)

```
import { DynamoDBClient, PutItemCommand } from "@aws-sdk/client-dynamodb";

export function createDynamoDBItem(table_name, region, partition_key, sort_key) {
    const client = new DynamoDBClient({
        region: region,
        endpoint: `https://dynamodb.${region}.amazonaws.com`
    });

    // Get the current time in epoch second format
    const current_time = Math.floor(new Date().getTime() / 1000);
```

```
    // Calculate the expireAt time (90 days from now) in epoch second format
    const expire_at = Math.floor((new Date().getTime() + 90 * 24 * 60 * 60 *
 1000) / 1000);

    // Create DynamoDB item
    const item = {
        'partitionKey': {'S': partition_key},
        'sortKey': {'S': sort_key},
        'createdAt': {'N': current_time.toString()},
        'expireAt': {'N': expire_at.toString()}
    };

    const putItemCommand = new PutItemCommand({
        TableName: table_name,
        Item: item,
        ProvisionedThroughput: {
            ReadCapacityUnits: 1,
            WriteCapacityUnits: 1,
        },
    });

    client.send(putItemCommand, function(err, data) {
        if (err) {
            console.log("Exception encountered when creating item %s, here's what
 happened: ", data, err);
            throw err;
        } else {
            console.log("Item created successfully: %s.", data);
            return data;
        }
    });
}

// Example usage (commented out for testing)
// createDynamoDBItem('your-table-name', 'us-east-1', 'your-partition-key-value',
 'your-sort-key-value');
```

- For API details, see PutItem in *AWS SDK for JavaScript API Reference*.

Python

SDK for Python (Boto3)

```
from datetime import datetime, timedelta

import boto3


def create_dynamodb_item(table_name, region, primary_key, sort_key):
    """
    Creates a DynamoDB item with an attached expiry attribute.

    :param table_name: Table name for the boto3 resource to target when creating
 an item
    :param region: string representing the AWS region. Example: `us-east-1`
    :param primary_key: one attribute known as the partition key.
    :param sort_key: Also known as a range attribute.
    :return: Void (nothing)
    """
    try:
        dynamodb = boto3.resource("dynamodb", region_name=region)
        table = dynamodb.Table(table_name)

        # Get the current time in epoch second format
        current_time = int(datetime.now().timestamp())

        # Calculate the expiration time (90 days from now) in epoch second format
        expiration_time = int((datetime.now() + timedelta(days=90)).timestamp())

        item = {
            "primaryKey": primary_key,
            "sortKey": sort_key,
            "creationDate": current_time,
            "expireAt": expiration_time,
        }
        response = table.put_item(Item=item)

        print("Item created successfully.")
        return response
    except Exception as e:
        print(f"Error creating item: {e}")
        raise e
```

```
# Use your own values
create_dynamodb_item(
    "your-table-name", "us-west-2", "your-partition-key-value", "your-sort-key-
value"
)
```

- For API details, see PutItem in *AWS SDK for Python (Boto3) API Reference*.

Update an item and refresh the Time to Live

This example is a continuation of the one from the previous section. The expiration time can be recomputed if the item is updated. The following example recomputes the `expireAt` timestamp to be 90 days from the current time.

The following code examples show how to update an item's TTL.

Java

SDK for Java 2.x

Update TTL on an existing DynamoDB item in a table.

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.ResourceNotFoundException;
import software.amazon.awssdk.services.dynamodb.model.UpdateItemRequest;
import software.amazon.awssdk.services.dynamodb.model.UpdateItemResponse;

import java.util.HashMap;
import java.util.Map;
import java.util.Optional;

    public UpdateItemResponse updateItemWithTTL(
        final String tableName, final String primaryKeyValue, final String
 sortKeyValue) {
        // Get current time in epoch second format
        final long currentTime = System.currentTimeMillis() / 1000;
```

```
        // Calculate expiration time 90 days from now in epoch second format
        final long expireDate = currentTime + (DAYS_TO_EXPIRE * SECONDS_PER_DAY);

        // Create the key map for the item to update
        final Map<String, AttributeValue> keyMap = new HashMap<>();
        keyMap.put(PRIMARY_KEY_ATTR,
AttributeValue.builder().s(primaryKeyValue).build());
        keyMap.put(SORT_KEY_ATTR,
AttributeValue.builder().s(sortKeyValue).build());

        // Create the expression attribute values
        final Map<String, AttributeValue> expressionAttributeValues = new
HashMap<>();
        expressionAttributeValues.put(
            ":c",
AttributeValue.builder().n(String.valueOf(currentTime)).build());
        expressionAttributeValues.put(
            ":e",
AttributeValue.builder().n(String.valueOf(expireDate)).build());

        final UpdateItemRequest request = UpdateItemRequest.builder()
            .tableName(tableName)
            .key(keyMap)
            .updateExpression(UPDATE_EXPRESSION)
            .expressionAttributeValues(expressionAttributeValues)
            .build();

        try {
            final UpdateItemResponse response =
dynamoDbClient.updateItem(request);
            System.out.println(String.format(SUCCESS_MESSAGE, tableName));
            return response;
        } catch (ResourceNotFoundException e) {
            System.err.format(TABLE_NOT_FOUND_ERROR, tableName);
            throw e;
        } catch (DynamoDbException e) {
            System.err.println(e.getMessage());
            throw e;
        }
    }
```

- For API details, see UpdateItem in *AWS SDK for Java 2.x API Reference*.

JavaScript

SDK for JavaScript (v3)

```
import { DynamoDBClient, UpdateItemCommand } from "@aws-sdk/client-dynamodb";
import { marshall, unmarshall } from "@aws-sdk/util-dynamodb";

export const updateItem = async (tableName, partitionKey, sortKey, region = 'us-
east-1') => {
    const client = new DynamoDBClient({
        region: region,
        endpoint: `https://dynamodb.${region}.amazonaws.com`
    });

    const currentTime = Math.floor(Date.now() / 1000);
    const expireAt = Math.floor((Date.now() + 90 * 24 * 60 * 60 * 1000) / 1000);

    const params = {
        TableName: tableName,
        Key: marshall({
            partitionKey: partitionKey,
            sortKey: sortKey
        }),
        UpdateExpression: "SET updatedAt = :c, expireAt = :e",
        ExpressionAttributeValues: marshall({
            ":c": currentTime,
            ":e": expireAt
        }),
    };

    try {
        const data = await client.send(new UpdateItemCommand(params));
        const responseData = unmarshall(data.Attributes);
        console.log("Item updated successfully: %s", responseData);
        return responseData;
    } catch (err) {
        console.error("Error updating item:", err);
        throw err;
    }
}

// Example usage (commented out for testing)
// updateItem('your-table-name', 'your-partition-key-value', 'your-sort-key-
value');
```

- For API details, see UpdateItem in *AWS SDK for JavaScript API Reference*.

Python

SDK for Python (Boto3)

```
from datetime import datetime, timedelta

import boto3


def update_dynamodb_item(table_name, region, primary_key, sort_key):
    """
    Update an existing DynamoDB item with a TTL.
    :param table_name: Name of the DynamoDB table
    :param region: AWS Region of the table - example `us-east-1`
    :param primary_key: one attribute known as the partition key.
    :param sort_key: Also known as a range attribute.
    :return: Void (nothing)
    """
    try:
        # Create the DynamoDB resource.
        dynamodb = boto3.resource("dynamodb", region_name=region)
        table = dynamodb.Table(table_name)

        # Get the current time in epoch second format
        current_time = int(datetime.now().timestamp())

        # Calculate the expireAt time (90 days from now) in epoch second format
        expire_at = int((datetime.now() + timedelta(days=90)).timestamp())

        table.update_item(
            Key={"partitionKey": primary_key, "sortKey": sort_key},
            UpdateExpression="set updatedAt=:c, expireAt=:e",
            ExpressionAttributeValues={":c": current_time, ":e": expire_at},
        )

        print("Item updated successfully.")
    except Exception as e:
        print(f"Error updating item: {e}")
```

```
# Replace with your own values
update_dynamodb_item(
    "your-table-name", "us-west-2", "your-partition-key-value", "your-sort-key-
value"
)
```

- For API details, see UpdateItem in *AWS SDK for Python (Boto3) API Reference*.

The TTL examples discussed in this introduction demonstrate a method to ensure only recently updated items are kept in a table. Updated items have their lifespan extended, whereas items not updated post-creation expire and are deleted at no cost, reducing storage and maintaining clean tables.

Working with expired items and time to live (TTL)

Expired items that are pending deletion can be filtered from read and write operations. This is useful in scenarios when expired data is no longer valid and should not be used. If they are not filtered, they'll continue to show in read and write operations until they are deleted by the background process.

> **Note**
>
> These items still count towards storage and read costs until they are deleted.

TTL deletions can be identified in DynamoDB Streams, but only in the Region where the deletion occurred. TTL deletions that are replicated to global table regions are not identifiable in DynamoDB streams in the regions the deletion is replicated to.

Filter expired items from read operations

For read operations such as Scan and Query, a filter expression can filter out expired items that are pending deletion. As shown in the following code snippet, the filter expression can filter out items where the TTL time is equal to or less than the current time. For example, the Python SDK code includes an assignment statement that obtains the current time as a variable (now), and converts it into `int` for epoch time format.

The following code examples show how to query for TTL items.

Java

SDK for Java 2.x

Query Filtered Expression to gather TTL items in a DynamoDB table using AWS SDK for Java 2.x.

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.QueryRequest;
import software.amazon.awssdk.services.dynamodb.model.QueryResponse;
import software.amazon.awssdk.services.dynamodb.model.ResourceNotFoundException;

import java.util.Map;
import java.util.Optional;

        final QueryRequest request = QueryRequest.builder()
            .tableName(tableName)
            .keyConditionExpression(KEY_CONDITION_EXPRESSION)
            .filterExpression(FILTER_EXPRESSION)
            .expressionAttributeNames(expressionAttributeNames)
            .expressionAttributeValues(expressionAttributeValues)
            .build();

        try (DynamoDbClient ddb = dynamoDbClient != null
            ? dynamoDbClient
            : DynamoDbClient.builder().region(region).build()) {
            final QueryResponse response = ddb.query(request);
            System.out.println("Query successful. Found " + response.count() + "
 items that have not expired yet.");

            // Print each item
            response.items().forEach(item -> {
                System.out.println("Item: " + item);
            });

            return 0;
        } catch (ResourceNotFoundException e) {
            System.err.format(TABLE_NOT_FOUND_ERROR, tableName);
            throw e;
```

```
        } catch (DynamoDbException e) {
            System.err.println(e.getMessage());
            throw e;
        }
```

- For API details, see Query in *AWS SDK for Java 2.x API Reference*.

JavaScript

SDK for JavaScript (v3)

Query Filtered Expression to gather TTL items in a DynamoDB table using AWS SDK for JavaScript.

```
import { DynamoDBClient, QueryCommand } from "@aws-sdk/client-dynamodb";
import { marshall, unmarshall } from "@aws-sdk/util-dynamodb";

export const queryFiltered = async (tableName, primaryKey, region = 'us-east-1')
 => {
    const client = new DynamoDBClient({
        region: region,
        endpoint: `https://dynamodb.${region}.amazonaws.com`
    });

    const currentTime = Math.floor(Date.now() / 1000);

    const params = {
        TableName: tableName,
        KeyConditionExpression: "#pk = :pk",
        FilterExpression: "#ea > :ea",
        ExpressionAttributeNames: {
            "#pk": "primaryKey",
            "#ea": "expireAt"
        },
        ExpressionAttributeValues: marshall({
            ":pk": primaryKey,
            ":ea": currentTime
        })
    };

    try {
        const { Items } = await client.send(new QueryCommand(params));
```

```
        Items.forEach(item => {
            console.log(unmarshall(item))
        });
        return Items;
    } catch (err) {
        console.error(`Error querying items: ${err}`);
        throw err;
    }
}

// Example usage (commented out for testing)
// queryFiltered('your-table-name', 'your-partition-key-value');
```

- For API details, see Query in *AWS SDK for JavaScript API Reference*.

Python

SDK for Python (Boto3)

Query Filtered Expression to gather TTL items in a DynamoDB table using AWS SDK for Python (Boto3).

```
from datetime import datetime

import boto3


def query_dynamodb_items(table_name, partition_key):
    """

    :param table_name: Name of the DynamoDB table
    :param partition_key:
    :return:
    """
    try:
        # Initialize a DynamoDB resource
        dynamodb = boto3.resource("dynamodb", region_name="us-east-1")

        # Specify your table
        table = dynamodb.Table(table_name)

        # Get the current time in epoch format
```

```
        current_time = int(datetime.now().timestamp())

        # Perform the query operation with a filter expression to exclude expired
 items
        # response = table.query(
        #
 KeyConditionExpression=boto3.dynamodb.conditions.Key('partitionKey').eq(partition_key),
        #
 FilterExpression=boto3.dynamodb.conditions.Attr('expireAt').gt(current_time)
        # )
        response = table.query(

 KeyConditionExpression=dynamodb.conditions.Key("partitionKey").eq(partition_key),

 FilterExpression=dynamodb.conditions.Attr("expireAt").gt(current_time),
        )

        # Print the items that are not expired
        for item in response["Items"]:
            print(item)

    except Exception as e:
        print(f"Error querying items: {e}")


# Call the function with your values
query_dynamodb_items("Music", "your-partition-key-value")
```

- For API details, see Query in *AWS SDK for Python (Boto3) API Reference*.

Conditionally write to expired items

A condition expression can be used to avoid writes against expired items. The code snippet below is a conditional update that checks whether the expiration time is greater than the current time. If true, the write operation will continue.

The following code examples show how to conditionally update an item's TTL.

Java

SDK for Java 2.x

Update TTL on on an existing DynamoDB Item in a table, with a condition.

```
package com.amazon.samplelib.ttl;

import com.amazon.samplelib.CodeSampleUtils;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import
 software.amazon.awssdk.services.dynamodb.model.ConditionalCheckFailedException;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.ResourceNotFoundException;
import software.amazon.awssdk.services.dynamodb.model.UpdateItemRequest;
import software.amazon.awssdk.services.dynamodb.model.UpdateItemResponse;

import java.util.Map;
import java.util.Optional;

/**
 * Updates an item in a DynamoDB table with TTL attributes using a conditional
 expression.
 * This class demonstrates how to conditionally update TTL expiration timestamps.
 */
public class UpdateTTLConditional {

    private static final String USAGE =
        """
            Usage:
                <tableName> <primaryKey> <sortKey> <region>
            Where:
                tableName - The Amazon DynamoDB table being queried.
                primaryKey - The name of the primary key. Also known as the hash
 or partition key.
                sortKey - The name of the sort key. Also known as the range
 attribute.
                region (optional) - The AWS region that the Amazon DynamoDB table
 is located in. (Default: us-east-1)
            """;
    private static final int DAYS_TO_EXPIRE = 90;
```

```
    private static final int SECONDS_PER_DAY = 24 * 60 * 60;
    private static final String PRIMARY_KEY_ATTR = "primaryKey";
    private static final String SORT_KEY_ATTR = "sortKey";
    private static final String UPDATED_AT_ATTR = "updatedAt";
    private static final String EXPIRE_AT_ATTR = "expireAt";
    private static final String UPDATE_EXPRESSION = "SET " + UPDATED_AT_ATTR +
"=:c, " + EXPIRE_AT_ATTR + "=:e";
    private static final String CONDITION_EXPRESSION = "attribute_exists(" +
PRIMARY_KEY_ATTR + ")";
    private static final String SUCCESS_MESSAGE = "%s UpdateItem operation with
TTL successful.";
    private static final String CONDITION_FAILED_MESSAGE = "Condition check
failed. Item does not exist.";
    private static final String TABLE_NOT_FOUND_ERROR = "Error: The Amazon
DynamoDB table \"%s\" can't be found.";

    private final DynamoDbClient dynamoDbClient;

    /**
     * Constructs an UpdateTTLConditional with a default DynamoDB client.
     */
    public UpdateTTLConditional() {
        this.dynamoDbClient = null;
    }

    /**
     * Constructs an UpdateTTLConditional with the specified DynamoDB client.
     *
     * @param dynamoDbClient The DynamoDB client to use
     */
    public UpdateTTLConditional(final DynamoDbClient dynamoDbClient) {
        this.dynamoDbClient = dynamoDbClient;
    }

    /**
     * Main method to demonstrate conditionally updating an item with TTL.
     *
     * @param args Command line arguments
     */
    public static void main(final String[] args) {
        try {
            int result = new UpdateTTLConditional().processArgs(args);
            System.exit(result);
        } catch (Exception e) {
```

```
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }

    /**
     * Process command line arguments and conditionally update an item with TTL.
     *
     * @param args Command line arguments
     * @return 0 if successful, non-zero otherwise
     * @throws ResourceNotFoundException If the table doesn't exist
     * @throws DynamoDbException If an error occurs during the operation
     * @throws IllegalArgumentException If arguments are invalid
     */
    public int processArgs(final String[] args) {
        // Argument validation (remove or replace this line when reusing this
code)
        CodeSampleUtils.validateArgs(args, new int[] {3, 4}, USAGE);

        final String tableName = args[0];
        final String primaryKey = args[1];
        final String sortKey = args[2];
        final Region region = Optional.ofNullable(args.length > 3 ? args[3] :
null)
            .map(Region::of)
            .orElse(Region.US_EAST_1);

        // Get current time in epoch second format
        final long currentTime = System.currentTimeMillis() / 1000;

        // Calculate expiration time 90 days from now in epoch second format
        final long expireDate = currentTime + (DAYS_TO_EXPIRE * SECONDS_PER_DAY);

        // Create the key map for the item to update
        final Map<String, AttributeValue> keyMap = Map.of(
            PRIMARY_KEY_ATTR, AttributeValue.builder().s(primaryKey).build(),
            SORT_KEY_ATTR, AttributeValue.builder().s(sortKey).build());

        // Create the expression attribute values
        final Map<String, AttributeValue> expressionAttributeValues = Map.of(
            ":c",
AttributeValue.builder().n(String.valueOf(currentTime)).build(),
            ":e",
AttributeValue.builder().n(String.valueOf(expireDate)).build());
```

```
        final UpdateItemRequest request = UpdateItemRequest.builder()
            .tableName(tableName)
            .key(keyMap)
            .updateExpression(UPDATE_EXPRESSION)
            .conditionExpression(CONDITION_EXPRESSION)
            .expressionAttributeValues(expressionAttributeValues)
            .build();

        try (DynamoDbClient ddb = dynamoDbClient != null
            ? dynamoDbClient
            : DynamoDbClient.builder().region(region).build()) {
            final UpdateItemResponse response = ddb.updateItem(request);
            System.out.println(String.format(SUCCESS_MESSAGE, tableName));
            return 0;
        } catch (ConditionalCheckFailedException e) {
            System.err.println(CONDITION_FAILED_MESSAGE);
            throw e;
        } catch (ResourceNotFoundException e) {
            System.err.format(TABLE_NOT_FOUND_ERROR, tableName);
            throw e;
        } catch (DynamoDbException e) {
            System.err.println(e.getMessage());
            throw e;
        }
    }
}
```

- For API details, see [UpdateItem](#) in *AWS SDK for Java 2.x API Reference*.

JavaScript

SDK for JavaScript (v3)

Update TTL on on an existing DynamoDB Item in a table, with a condition.

```
import { DynamoDBClient, UpdateItemCommand } from "@aws-sdk/client-dynamodb";
import { marshall, unmarshall } from "@aws-sdk/util-dynamodb";

export const updateItemConditional = async (tableName, partitionKey, sortKey,
 region = 'us-east-1', newAttribute = 'default-value') => {
    const client = new DynamoDBClient({
```

```
        region: region,
        endpoint: `https://dynamodb.${region}.amazonaws.com`
    });

    const currentTime = Math.floor(Date.now() / 1000);

    const params = {
        TableName: tableName,
        Key: marshall({
            artist: partitionKey,
            album: sortKey
        }),
        UpdateExpression: "SET newAttribute = :newAttribute",
        ConditionExpression: "expireAt > :expiration",
        ExpressionAttributeValues: marshall({
            ':newAttribute': newAttribute,
            ':expiration': currentTime
        }),
        ReturnValues: "ALL_NEW"
    };

    try {
        const response = await client.send(new UpdateItemCommand(params));
        const responseData = unmarshall(response.Attributes);
        console.log("Item updated successfully: ", responseData);
        return responseData;
    } catch (error) {
        if (error.name === "ConditionalCheckFailedException") {
            console.log("Condition check failed: Item's 'expireAt' is expired.");
        } else {
            console.error("Error updating item: ", error);
        }
        throw error;
    }
};

// Example usage (commented out for testing)
// updateItemConditional('your-table-name', 'your-partition-key-value', 'your-
sort-key-value');
```

- For API details, see UpdateItem in *AWS SDK for JavaScript API Reference*.

Python

SDK for Python (Boto3)

Update TTL on on an existing DynamoDB Item in a table, with a condition.

```
from datetime import datetime, timedelta

import boto3
from botocore.exceptions import ClientError


def update_dynamodb_item_ttl(table_name, region, primary_key, sort_key,
 ttl_attribute):
    """
    Updates an existing record in a DynamoDB table with a new or updated TTL
 attribute.

    :param table_name: Name of the DynamoDB table
    :param region: AWS Region of the table - example `us-east-1`
    :param primary_key: one attribute known as the partition key.
    :param sort_key: Also known as a range attribute.
    :param ttl_attribute: name of the TTL attribute in the target DynamoDB table
    :return:
    """
    try:
        dynamodb = boto3.resource("dynamodb", region_name=region)
        table = dynamodb.Table(table_name)

        # Generate updated TTL in epoch second format
        updated_expiration_time = int((datetime.now() +
 timedelta(days=90)).timestamp())

        # Define the update expression for adding/updating a new attribute
        update_expression = "SET newAttribute = :val1"

        # Define the condition expression for checking if 'expireAt' is not
 expired
        condition_expression = "expireAt > :val2"

        # Define the expression attribute values
        expression_attribute_values = {":val1": ttl_attribute, ":val2":
 updated_expiration_time}
```

```
        response = table.update_item(
            Key={"primaryKey": primary_key, "sortKey": sort_key},
            UpdateExpression=update_expression,
            ConditionExpression=condition_expression,
            ExpressionAttributeValues=expression_attribute_values,
        )

        print("Item updated successfully.")
        return response["ResponseMetadata"]["HTTPStatusCode"]  # Ideally a 200 OK
    except ClientError as e:
        if e.response["Error"]["Code"] == "ConditionalCheckFailedException":
            print("Condition check failed: Item's 'expireAt' is expired.")
        else:
            print(f"Error updating item: {e}")
    except Exception as e:
        print(f"Error updating item: {e}")


# replace with your values
update_dynamodb_item_ttl(
    "your-table-name",
    "us-east-1",
    "your-partition-key-value",
    "your-sort-key-value",
    "your-ttl-attribute-value",
)
```

- For API details, see UpdateItem in *AWS SDK for Python (Boto3) API Reference*.

Identifying deleted items in DynamoDB Streams

The streams record contains a user identity field `Records[<index>].userIdentity`. Items that are deleted by the TTL process have the following fields:

```
Records[<index>].userIdentity.type
"Service"

Records[<index>].userIdentity.principalId
"dynamodb.amazonaws.com"
```

The following JSON shows the relevant portion of a single streams record:

```
"Records": [
  {
 ...
  "userIdentity": {
  "type": "Service",
       "principalId": "dynamodb.amazonaws.com"
    }
   ...
 }
]
```

Querying tables in DynamoDB

You can use the `Query` API operation in Amazon DynamoDB to find items based on primary key values.

You must provide the name of the partition key attribute and a single value for that attribute. `Query` returns all items with that partition key value. Optionally, you can provide a sort key attribute and use a comparison operator to refine the search results.

For more information on how to use `Query`, such as the request syntax, response parameters, and additional examples, see Query in the *Amazon DynamoDB API Reference*.

Topics

- Key condition expressions for the Query operation in DynamoDB
- Filter expressions for the Query operation in DynamoDB
- Paginating table query results in DynamoDB
- Other aspects of working with the Query operation in DynamoDB

Key condition expressions for the Query operation in DynamoDB

You can use any attribute name in a key condition expression, provided that the first character is `a-z` or `A-Z` and the rest of the characters (starting from the second character, if present) are `a-z`, `A-Z`, or `0-9`. In addition, the attribute name must not be a DynamoDB reserved word. (For a complete list of these, see Reserved words in DynamoDB.) If an attribute name does not meet these requirements, you must define an expression attribute name as a placeholder. For more information, see Expression attribute names (aliases) in DynamoDB.

For items with a given partition key value, DynamoDB stores these items close together, in sorted order by sort key value. In a `Query` operation, DynamoDB retrieves the items in sorted order, and then processes the items using `KeyConditionExpression` and any `FilterExpression` that might be present. Only then are the `Query` results sent back to the client.

A `Query` operation always returns a result set. If no matching items are found, the result set is empty.

`Query` results are always sorted by the sort key value. If the data type of the sort key is `Number`, the results are returned in numeric order. Otherwise, the results are returned in order of UTF-8 bytes. By default, the sort order is ascending. To reverse the order, set the `ScanIndexForward` parameter to `false`.

A single `Query` operation can retrieve a maximum of 1 MB of data. This limit applies before any `FilterExpression` or `ProjectionExpression` is applied to the results. If `LastEvaluatedKey` is present in the response and is non-null, you must paginate the result set (see Paginating table query results in DynamoDB).

Key condition expression examples

To specify the search criteria, you use a *key condition expression*—a string that determines the items to be read from the table or index.

You must specify the partition key name and value as an equality condition. You cannot use a non-key attribute in a key condition expression.

You can optionally provide a second condition for the sort key (if present). The sort key condition must use one of the following comparison operators:

- *a* = *b* — true if the attribute *a* is equal to the value *b*
- *a* < *b* — true if *a* is less than *b*
- *a* <= *b* — true if *a* is less than or equal to *b*
- *a* > *b* — true if *a* is greater than *b*
- *a* >= *b* — true if *a* is greater than or equal to *b*
- *a* `BETWEEN` *b* `AND` *c* — true if *a* is greater than or equal to *b*, and less than or equal to *c*.

The following function is also supported:

- `begins_with (a, substr)`— true if the value of attribute `a` begins with a particular substring.

The following AWS Command Line Interface (AWS CLI) examples demonstrate the use of key condition expressions. These expressions use placeholders (such as `:name` and `:sub`) instead of actual values. For more information, see Expression attribute names (aliases) in DynamoDB and Using expression attribute values in DynamoDB.

Example

Query the `Thread` table for a particular `ForumName` (partition key). All of the items with that `ForumName` value are read by the query because the sort key (`Subject`) is not included in `KeyConditionExpression`.

```
aws dynamodb query \
    --table-name Thread \
    --key-condition-expression "ForumName = :name" \
    --expression-attribute-values  '{":name":{"S":"Amazon DynamoDB"}}'
```

Example

Query the `Thread` table for a particular `ForumName` (partition key), but this time return only the items with a given `Subject` (sort key).

```
aws dynamodb query \
    --table-name Thread \
    --key-condition-expression "ForumName = :name and Subject = :sub" \
    --expression-attribute-values  file://values.json
```

The arguments for `--expression-attribute-values` are stored in the `values.json` file.

```
{
    ":name":{"S":"Amazon DynamoDB"},
    ":sub":{"S":"DynamoDB Thread 1"}
}
```

Example

Query the `Reply` table for a particular `Id` (partition key), but return only those items whose `ReplyDateTime` (sort key) begins with certain characters.

```
aws dynamodb query \
    --table-name Reply \
    --key-condition-expression "Id = :id and begins_with(ReplyDateTime, :dt)" \
    --expression-attribute-values  file://values.json
```

The arguments for `--expression-attribute-values` are stored in the `values.json` file.

```
{
    ":id":{"S":"Amazon DynamoDB#DynamoDB Thread 1"},
    ":dt":{"S":"2015-09"}
}
```

Filter expressions for the Query operation in DynamoDB

If you need to further refine the `Query` results, you can optionally provide a filter expression. A *filter expression* determines which items within the `Query` results should be returned to you. All of the other results are discarded.

A filter expression is applied after a `Query` finishes, but before the results are returned. Therefore, a `Query` consumes the same amount of read capacity, regardless of whether a filter expression is present.

A `Query` operation can retrieve a maximum of 1 MB of data. This limit applies before the filter expression is evaluated.

A filter expression cannot contain partition key or sort key attributes. You need to specify those attributes in the key condition expression, not the filter expression.

The syntax for a filter expression is similar to that of a key condition expression. Filter expressions can use the same comparators, functions, and logical operators as a key condition expression. In addition, filter expressions can use the not-equals operator (<>), the `OR` operator, the `CONTAINS` operator, the `IN` operator, the `BEGINS_WITH` operator, the `BETWEEN` operator, the `EXISTS` operator, and the `SIZE` operator. For more information, see Key condition expressions for the Query operation in DynamoDB and Syntax for filter and condition expressions.

Example

The following AWS CLI example queries the `Thread` table for a particular `ForumName` (partition key) and `Subject` (sort key). Of the items that are found, only the most popular discussion threads are returned—in other words, only those threads with more than a certain number of `Views`.

```
aws dynamodb query \
    --table-name Thread \
    --key-condition-expression "ForumName = :fn and Subject begins_with :sub" \
    --filter-expression "#v >= :num" \
    --expression-attribute-names '{"#v": "Views"}' \
    --expression-attribute-values file://values.json
```

The arguments for `--expression-attribute-values` are stored in the `values.json` file.

```
{
    ":fn":{"S":"Amazon DynamoDB"},
    ":sub":{"S":"DynamoDB Thread 1"},
    ":num":{"N":"3"}
}
```

Note that `Views` is a reserved word in DynamoDB (see Reserved words in DynamoDB), so this example uses `#v` as a placeholder. For more information, see Expression attribute names (aliases) in DynamoDB.

> **Note**
>
> A filter expression removes items from the `Query` result set. If possible, avoid using `Query` where you expect to retrieve a large number of items but also need to discard most of those items.

Paginating table query results in DynamoDB

DynamoDB *paginates* the results from `Query` operations. With pagination, the `Query` results are divided into "pages" of data that are 1 MB in size (or less). An application can process the first page of results, then the second page, and so on.

A single `Query` only returns a result set that fits within the 1 MB size limit. To determine whether there are more results, and to retrieve them one page at a time, applications should do the following:

1. Examine the low-level `Query` result:
   - If the result contains a `LastEvaluatedKey` element and it's non-null, proceed to step 2.
   - If there is *not* a `LastEvaluatedKey` in the result, there are no more items to be retrieved.

2. Construct a `Query` with the same `KeyConditionExpression`. However, this time, take the `LastEvaluatedKey` value from step 1 and use it as the `ExclusiveStartKey` parameter in the new `Query` request.
3. Run the new `Query` request.
4. Go to step 1.

In other words, the `LastEvaluatedKey` from a `Query` response should be used as the `ExclusiveStartKey` for the next `Query` request. If there is not a `LastEvaluatedKey` element in a `Query` response, then you have retrieved the final page of results. If `LastEvaluatedKey` is not empty, it does not necessarily mean that there is more data in the result set. The only way to know when you have reached the end of the result set is when `LastEvaluatedKey` is empty.

You can use the AWS CLI to view this behavior. The AWS CLI sends low-level `Query` requests to DynamoDB repeatedly, until `LastEvaluatedKey` is no longer present in the results. Consider the following AWS CLI example that retrieves movie titles from a particular year.

```
aws dynamodb query --table-name Movies \
    --projection-expression "title" \
    --key-condition-expression "#y = :yyyy" \
    --expression-attribute-names '{"#y":"year"}' \
    --expression-attribute-values '{":yyyy":{"N":"1993"}}' \
    --page-size 5 \
    --debug
```

Ordinarily, the AWS CLI handles pagination automatically. However, in this example, the AWS CLI `--page-size` parameter limits the number of items per page. The `--debug` parameter prints low-level information about requests and responses.

If you run the example, the first response from DynamoDB looks similar to the following.

```
2017-07-07 11:13:15,603 - MainThread - botocore.parsers - DEBUG - Response body:
b'{"Count":5,"Items":[{"title":{"S":"A Bronx Tale"}},
{"title":{"S":"A Perfect World"}},{"title":{"S":"Addams Family Values"}},
{"title":{"S":"Alive"}},{"title":{"S":"Benny & Joon"}}],
"LastEvaluatedKey":{"year":{"N":"1993"},"title":{"S":"Benny & Joon"}},
"ScannedCount":5}'
```

The `LastEvaluatedKey` in the response indicates that not all of the items have been retrieved. The AWS CLI then issues another `Query` request to DynamoDB. This request and response pattern continues, until the final response.

```
2017-07-07 11:13:16,291 - MainThread - botocore.parsers - DEBUG - Response body:
b'{"Count":1,"Items":[{"title":{"S":"What\'s Eating Gilbert
 Grape"}}],"ScannedCount":1}'
```

The absence of `LastEvaluatedKey` indicates that there are no more items to retrieve.

> **Note**
>
> The AWS SDKs handle the low-level DynamoDB responses (including the presence or absence of `LastEvaluatedKey`) and provide various abstractions for paginating `Query` results. For example, the SDK for Java document interface provides `java.util.Iterator` support so that you can walk through the results one at a time.
> For code examples in various programming languages, see the Amazon DynamoDB Getting Started Guide and the AWS SDK documentation for your language.

You can also reduce page size by limiting the number of items in the result set, with the `Limit` parameter of the `Query` operation.

For more information about querying with DynamoDB, see Querying tables in DynamoDB.

Other aspects of working with the Query operation in DynamoDB

This section covers additional aspects of the DynamoDB Query operation, including limiting result size, counting scanned vs. returned items, monitoring read capacity consumption, and controlling read consistency.

Limiting the number of items in the result set

With the `Query` operation, you can limit the number of items that it reads. To do this, set the `Limit` parameter to the maximum number of items that you want.

For example, suppose that you `Query` a table, with a `Limit` value of 6, and without a filter expression. The `Query` result contains the first six items from the table that match the key condition expression from the request.

Now suppose that you add a filter expression to the `Query`. In this case, DynamoDB reads up to six items, and then returns only those that match the filter expression. The final `Query` result contains six items or fewer, even if more items would have matched the filter expression if DynamoDB had kept reading more items.

Counting the items in the results

In addition to the items that match your criteria, the `Query` response contains the following elements:

- `ScannedCount` — The number of items that matched the key condition expression *before* a filter expression (if present) was applied.
- `Count` — The number of items that remain *after* a filter expression (if present) was applied.

> **Note**
>
> If you don't use a filter expression, `ScannedCount` and `Count` have the same value.

If the size of the `Query` result set is larger than 1 MB, `ScannedCount` and `Count` represent only a partial count of the total items. You need to perform multiple `Query` operations to retrieve all the results (see Paginating table query results in DynamoDB).

Each `Query` response contains the `ScannedCount` and `Count` for the items that were processed by that particular `Query` request. To obtain grand totals for all of the `Query` requests, you could keep a running tally of both `ScannedCount` and `Count`.

Capacity units consumed by query

You can `Query` any table or secondary index, as long as you provide the name of the partition key attribute and a single value for that attribute. `Query` returns all items with that partition key value. Optionally, you can provide a sort key attribute and use a comparison operator to refine the search results. `Query` API operations consume read capacity units, as follows.

If you `Query` a...	DynamoDB consumes read capacity units from...
Table	The table's provisioned read capacity.

If you Query a...	DynamoDB consumes read capacity units from...
Global secondary index	The index's provisioned read capacity.
Local secondary index	The base table's provisioned read capacity.

By default, a Query operation does not return any data on how much read capacity it consumes. However, you can specify the ReturnConsumedCapacity parameter in a Query request to obtain this information. The following are the valid settings for ReturnConsumedCapacity:

- NONE — No consumed capacity data is returned. (This is the default.)
- TOTAL — The response includes the aggregate number of read capacity units consumed.
- INDEXES — The response shows the aggregate number of read capacity units consumed, together with the consumed capacity for each table and index that was accessed.

DynamoDB calculates the number of read capacity units consumed based on the number of items and the size of those items, not on the amount of data that is returned to an application. For this reason, the number of capacity units consumed is the same whether you request all of the attributes (the default behavior) or just some of them (using a projection expression). The number is also the same whether or not you use a filter expression. Query consumes a minimum read capacity unit to perform one strongly consistent read per second, or two eventually consistent reads per second for an item up to 4 KB. If you need to read an item that is larger than 4 KB, DynamoDB needs additional read request units. Empty tables and very large tables which have a sparse amount of partition keys might see some additional RCUs charged beyond the amount of data queried. This covers the cost of serving the Query request, even if no data exists.

Read consistency for query

A Query operation performs eventually consistent reads, by default. This means that the Query results might not reflect changes due to recently completed PutItem or UpdateItem operations. For more information, see DynamoDB read consistency.

If you require strongly consistent reads, set the ConsistentRead parameter to true in the Query request.

Scanning tables in DynamoDB

A `Scan` operation in Amazon DynamoDB reads every item in a table or a secondary index. By default, a `Scan` operation returns all of the data attributes for every item in the table or index. You can use the `ProjectionExpression` parameter so that `Scan` only returns some of the attributes, rather than all of them.

`Scan` always returns a result set. If no matching items are found, the result set is empty.

A single `Scan` request can retrieve a maximum of 1 MB of data. Optionally, DynamoDB can apply a filter expression to this data, narrowing the results before they are returned to the user.

Topics

- Filter expressions for scan
- Limiting the number of items in the result set
- Paginating the results
- Counting the items in the results
- Capacity units consumed by scan
- Read consistency for scan
- Parallel scan

Filter expressions for scan

If you need to further refine the `Scan` results, you can optionally provide a filter expression. A *filter expression* determines which items within the `Scan` results should be returned to you. All of the other results are discarded.

A filter expression is applied after a `Scan` finishes but before the results are returned. Therefore, a `Scan` consumes the same amount of read capacity, regardless of whether a filter expression is present.

A `Scan` operation can retrieve a maximum of 1 MB of data. This limit applies before the filter expression is evaluated.

With `Scan`, you can specify any attributes in a filter expression—including partition key and sort key attributes.

The syntax for a filter expression is identical to that of a condition expression. Filter expressions can use the same comparators, functions, and logical operators as a condition expression. See Condition and filter expressions, operators, and functions in DynamoDB for more information about logical operators.

Example

The following AWS Command Line Interface (AWS CLI) example scans the `Thread` table and returns only the items that were last posted to by a particular user.

```
aws dynamodb scan \
     --table-name Thread \
     --filter-expression "LastPostedBy = :name" \
     --expression-attribute-values '{":name":{"S":"User A"}}'
```

Limiting the number of items in the result set

The `Scan` operation enables you to limit the number of items that it returns in the result. To do this, set the `Limit` parameter to the maximum number of items that you want the `Scan` operation to return, prior to filter expression evaluation.

For example, suppose that you `Scan` a table with a `Limit` value of 6 and without a filter expression. The `Scan` result contains the first six items from the table.

Now suppose that you add a filter expression to the `Scan`. In this case, DynamoDB applies the filter expression to the six items that were returned, discarding those that do not match. The final `Scan` result contains six items or fewer, depending on the number of items that were filtered.

Paginating the results

DynamoDB *paginates* the results from `Scan` operations. With pagination, the `Scan` results are divided into "pages" of data that are 1 MB in size (or less). An application can process the first page of results, then the second page, and so on.

A single `Scan` only returns a result set that fits within the 1 MB size limit.

To determine whether there are more results and to retrieve them one page at a time, applications should do the following:

1. Examine the low-level `Scan` result:

   - If the result contains a `LastEvaluatedKey` element, proceed to step 2.

- If there is *not* a `LastEvaluatedKey` in the result, then there are no more items to be retrieved.

2. Construct a new `Scan` request, with the same parameters as the previous one. However, this time, take the `LastEvaluatedKey` value from step 1 and use it as the `ExclusiveStartKey` parameter in the new `Scan` request.
3. Run the new `Scan` request.
4. Go to step 1.

In other words, the `LastEvaluatedKey` from a `Scan` response should be used as the `ExclusiveStartKey` for the next `Scan` request. If there is not a `LastEvaluatedKey` element in a `Scan` response, you have retrieved the final page of results. (The absence of `LastEvaluatedKey` is the only way to know that you have reached the end of the result set.)

You can use the AWS CLI to view this behavior. The AWS CLI sends low-level `Scan` requests to DynamoDB, repeatedly, until `LastEvaluatedKey` is no longer present in the results. Consider the following AWS CLI example that scans the entire `Movies` table but returns only the movies from a particular genre.

```
aws dynamodb scan \
    --table-name Movies \
    --projection-expression "title" \
    --filter-expression 'contains(info.genres,:gen)' \
    --expression-attribute-values '{":gen":{"S":"Sci-Fi"}}' \
    --page-size 100  \
    --debug
```

Ordinarily, the AWS CLI handles pagination automatically. However, in this example, the AWS CLI `--page-size` parameter limits the number of items per page. The `--debug` parameter prints low-level information about requests and responses.

> **Note**
>
> Your pagination results will also differ based on the input parameters you pass.
>
> - Using `aws dynamodb scan --table-name Prices --max-items 1` returns a `NextToken`
> - Using `aws dynamodb scan --table-name Prices --limit 1` returns a `LastEvaluatedKey`.

Also be aware that using `--starting-token` in particular requires the `NextToken` value.

If you run the example, the first response from DynamoDB looks similar to the following.

```
2017-07-07 12:19:14,389 - MainThread - botocore.parsers - DEBUG - Response body:
b'{"Count":7,"Items":[{"title":{"S":"Monster on the Campus"}},{"title":{"S":"+1"}},
{"title":{"S":"100 Degrees Below Zero"}},{"title":{"S":"About Time"}},{"title":
{"S":"After Earth"}},
{"title":{"S":"Age of Dinosaurs"}},{"title":{"S":"Cloudy with a Chance of Meatballs
 2"}}],
"LastEvaluatedKey":{"year":{"N":"2013"},"title":{"S":"Curse of
 Chucky"}},"ScannedCount":100}'
```

The `LastEvaluatedKey` in the response indicates that not all of the items have been retrieved. The AWS CLI then issues another `Scan` request to DynamoDB. This request and response pattern continues, until the final response.

```
2017-07-07 12:19:17,830 - MainThread - botocore.parsers - DEBUG - Response body:
b'{"Count":1,"Items":[{"title":{"S":"WarGames"}}],"ScannedCount":6}'
```

The absence of `LastEvaluatedKey` indicates that there are no more items to retrieve.

Note

The AWS SDKs handle the low-level DynamoDB responses (including the presence or absence of `LastEvaluatedKey`) and provide various abstractions for paginating `Scan` results. For example, the SDK for Java document interface provides `java.util.Iterator` support so that you can walk through the results one at a time.
For code examples in various programming languages, see the Amazon DynamoDB Getting Started Guide and the AWS SDK documentation for your language.

Counting the items in the results

In addition to the items that match your criteria, the `Scan` response contains the following elements:

- `ScannedCount` — The number of items evaluated, before any `ScanFilter` is applied. A high `ScannedCount` value with few, or no, `Count` results indicates an inefficient `Scan` operation. If you did not use a filter in the request, `ScannedCount` is the same as `Count`.
- `Count` — The number of items that remain, *after* a filter expression (if present) was applied.

> **Note**
>
> If you do not use a filter expression, `ScannedCount` and `Count` have the same value.

If the size of the `Scan` result set is larger than 1 MB, `ScannedCount` and `Count` represent only a partial count of the total items. You need to perform multiple `Scan` operations to retrieve all the results (see Paginating the results).

Each `Scan` response contains the `ScannedCount` and `Count` for the items that were processed by that particular `Scan` request. To get grand totals for all of the `Scan` requests, you could keep a running tally of both `ScannedCount` and `Count`.

Capacity units consumed by scan

You can `Scan` any table or secondary index. `Scan` operations consume read capacity units, as follows.

If you Scan a...	DynamoDB consumes read capacity units from...
Table	The table's provisioned read capacity.
Global secondary index	The index's provisioned read capacity.
Local secondary index	The base table's provisioned read capacity.

> **Note**
>
> Cross-account access for secondary index scan operations is currently not supported with resource-based policies.

By default, a Scan operation does not return any data on how much read capacity it consumes. However, you can specify the ReturnConsumedCapacity parameter in a Scan request to obtain this information. The following are the valid settings for ReturnConsumedCapacity:

- NONE — No consumed capacity data is returned. (This is the default.)
- TOTAL — The response includes the aggregate number of read capacity units consumed.
- INDEXES — The response shows the aggregate number of read capacity units consumed, together with the consumed capacity for each table and index that was accessed.

DynamoDB calculates the number of read capacity units consumed based on the number of items and the size of those items, not on the amount of data that is returned to an application. For this reason, the number of capacity units consumed is the same whether you request all of the attributes (the default behavior) or just some of them (using a projection expression). The number is also the same whether or not you use a filter expression. Scan consumes a minimum read capacity unit to perform one strongly consistent read per second, or two eventually consistent reads per second for an item up to 4 KB. If you need to read an item that is larger than 4 KB, DynamoDB needs additional read request units. Empty tables and very large tables which have a sparse amount of partition keys might see some additional RCUs charged beyond the amount of data scanned. This covers the cost of serving the Scan request, even if no data exists.

Read consistency for scan

A Scan operation performs eventually consistent reads, by default. This means that the Scan results might not reflect changes due to recently completed PutItem or UpdateItem operations. For more information, see DynamoDB read consistency.

If you require strongly consistent reads, as of the time that the Scan begins, set the ConsistentRead parameter to true in the Scan request. This ensures that all of the write operations that completed before the Scan began are included in the Scan response.

Setting ConsistentRead to true can be useful in table backup or replication scenarios, in conjunction with DynamoDB Streams. You first use Scan with ConsistentRead set to true to obtain a consistent copy of the data in the table. During the Scan, DynamoDB Streams records any additional write activity that occurs on the table. After the Scan is complete, you can apply the write activity from the stream to the table.

> **Note**
>
> A `Scan` operation with `ConsistentRead` set to `true` consumes twice as many read capacity units as compared to leaving `ConsistentRead` at its default value (`false`).

Parallel scan

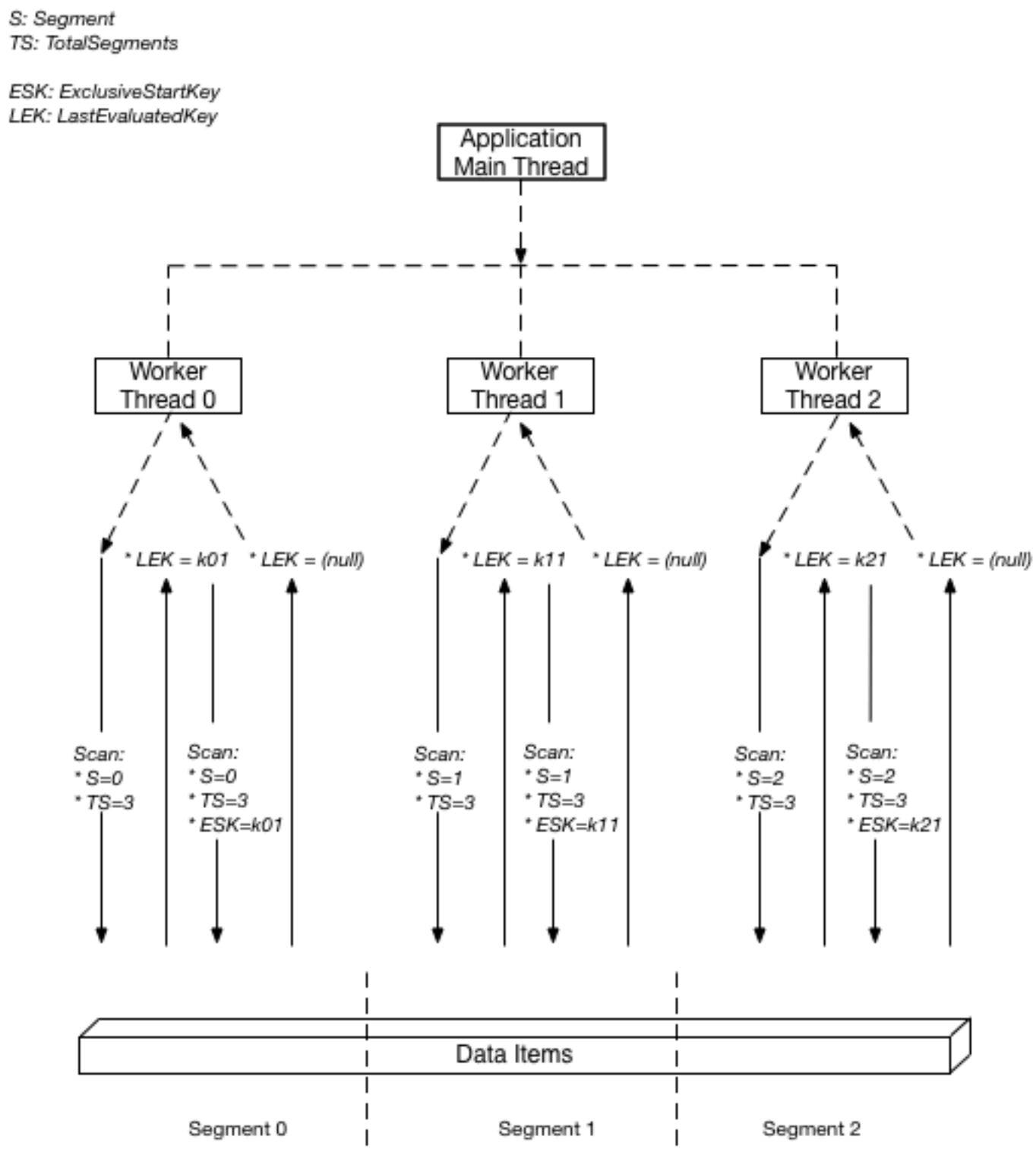
By default, the `Scan` operation processes data sequentially. Amazon DynamoDB returns data to the application in 1 MB increments, and an application performs additional `Scan` operations to retrieve the next 1 MB of data.

The larger the table or index being scanned, the more time the `Scan` takes to complete. In addition, a sequential `Scan` might not always be able to fully use the provisioned read throughput capacity: Even though DynamoDB distributes a large table's data across multiple physical partitions, a `Scan` operation can only read one partition at a time. For this reason, the throughput of a `Scan` is constrained by the maximum throughput of a single partition.

To address these issues, the `Scan` operation can logically divide a table or secondary index into multiple *segments*, with multiple application workers scanning the segments in parallel. Each worker can be a thread (in programming languages that support multithreading) or an operating system process. To perform a parallel scan, each worker issues its own `Scan` request with the following parameters:

- `Segment` — A segment to be scanned by a particular worker. Each worker should use a different value for `Segment`.
- `TotalSegments` — The total number of segments for the parallel scan. This value must be the same as the number of workers that your application will use.

The following diagram shows how a multithreaded application performs a parallel `Scan` with three degrees of parallelism.



In this diagram, the application spawns three threads and assigns each thread a number. (Segments are zero-based, so the first number is always 0.) Each thread issues a `Scan` request, setting `Segment` to its designated number and setting `TotalSegments` to 3. Each thread scans its

designated segment, retrieving data 1 MB at a time, and returns the data to the application's main thread.

DynamoDB assigns items to *segments* by applying a hash function to each item's partition key. For a given `TotalSegments` value, all items with the same partition key are always assigned to the same `Segment`. This means that in a table where *Item 1*, *Item 2*, and *Item 3* all share `pk="account#123"` (but have different sort keys), these items will be processed by the same worker, regardless of the sort key values or the size of the *item collection*.

Because *segment* assignment is based solely on the partition key hash, segments can be unevenly distributed. Some segments might contain no items, while others might contain many partition keys with large item collections. As a result, increasing the total number of segments does not guarantee faster scan performance, particularly when partition keys are not uniformly distributed across the keyspace.

The values for `Segment` and `TotalSegments` apply to individual `Scan` requests, and you can use different values at any time. You might need to experiment with these values, and the number of workers you use, until your application achieves its best performance.

> **Note**
>
> A parallel scan with a large number of workers can easily consume all of the provisioned throughput for the table or index being scanned. It is best to avoid such scans if the table or index is also incurring heavy read or write activity from other applications.
> To control the amount of data returned per request, use the `Limit` parameter. This can help prevent situations where one worker consumes all of the provisioned throughput, at the expense of all other workers.

PartiQL - a SQL-compatible query language for Amazon DynamoDB

Amazon DynamoDB supports PartiQL, a SQL-compatible query language, to select, insert, update, and delete data in Amazon DynamoDB. Using PartiQL, you can easily interact with DynamoDB tables and run ad hoc queries using the AWS Management Console, NoSQL Workbench, AWS Command Line Interface, and DynamoDB APIs for PartiQL.

PartiQL operations provide the same availability, latency, and performance as the other DynamoDB data plane operations.

The following sections describe the DynamoDB implementation of PartiQL.

Topics

- What is PartiQL?
- PartiQL in Amazon DynamoDB
- Getting started with PartiQL for DynamoDB
- PartiQL data types for DynamoDB
- PartiQL statements for DynamoDB
- Use PartiQL functions with DynamoDB
- PartiQL arithmetic, comparison, and logical operators for DynamoDB
- Performing transactions with PartiQL for DynamoDB
- Running batch operations with PartiQL for DynamoDB
- IAM security policies with PartiQL for DynamoDB

What is PartiQL?

PartiQL provides SQL-compatible query access across multiple data stores containing structured data, semistructured data, and nested data. It is widely used within Amazon and is now available as part of many AWS services, including DynamoDB.

For the PartiQL specification and a tutorial on the core query language, see the PartiQL documentation.

> **Note**
>
> - Amazon DynamoDB supports a *subset* of the PartiQL query language.
> - Amazon DynamoDB does not support the Amazon ion data format or Amazon Ion literals.

PartiQL in Amazon DynamoDB

To run PartiQL queries in DynamoDB, you can use:

- The DynamoDB console
- The NoSQL Workbench

- The AWS Command Line Interface (AWS CLI)
- The DynamoDB APIs

For information about using these methods to access DynamoDB, see Accessing DynamoDB.

Getting started with PartiQL for DynamoDB

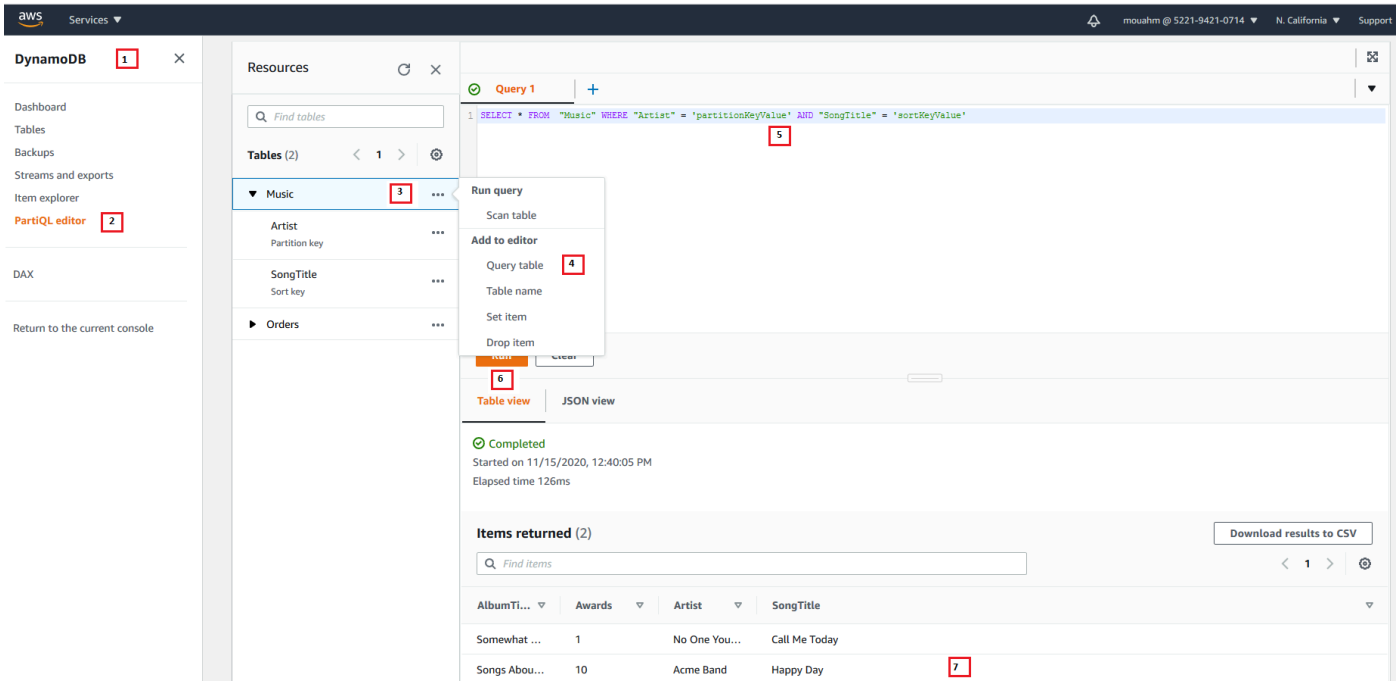
This section describes how to use PartiQL for DynamoDB from the Amazon DynamoDB console, the AWS Command Line Interface (AWS CLI), and DynamoDB APIs.

In the following examples, the DynamoDB table that is defined in the Getting started with DynamoDB tutorial is a pre-requisite.

For information about using the DynamoDB console, AWS Command Line Interface, or DynamoDB APIs to access DynamoDB, see Accessing DynamoDB.

To download and use the NoSQL workbench to build PartiQL for DynamoDB statements choose **PartiQL operations** at the top right corner of the NoSQL Workbench for DynamoDB Operation builder.

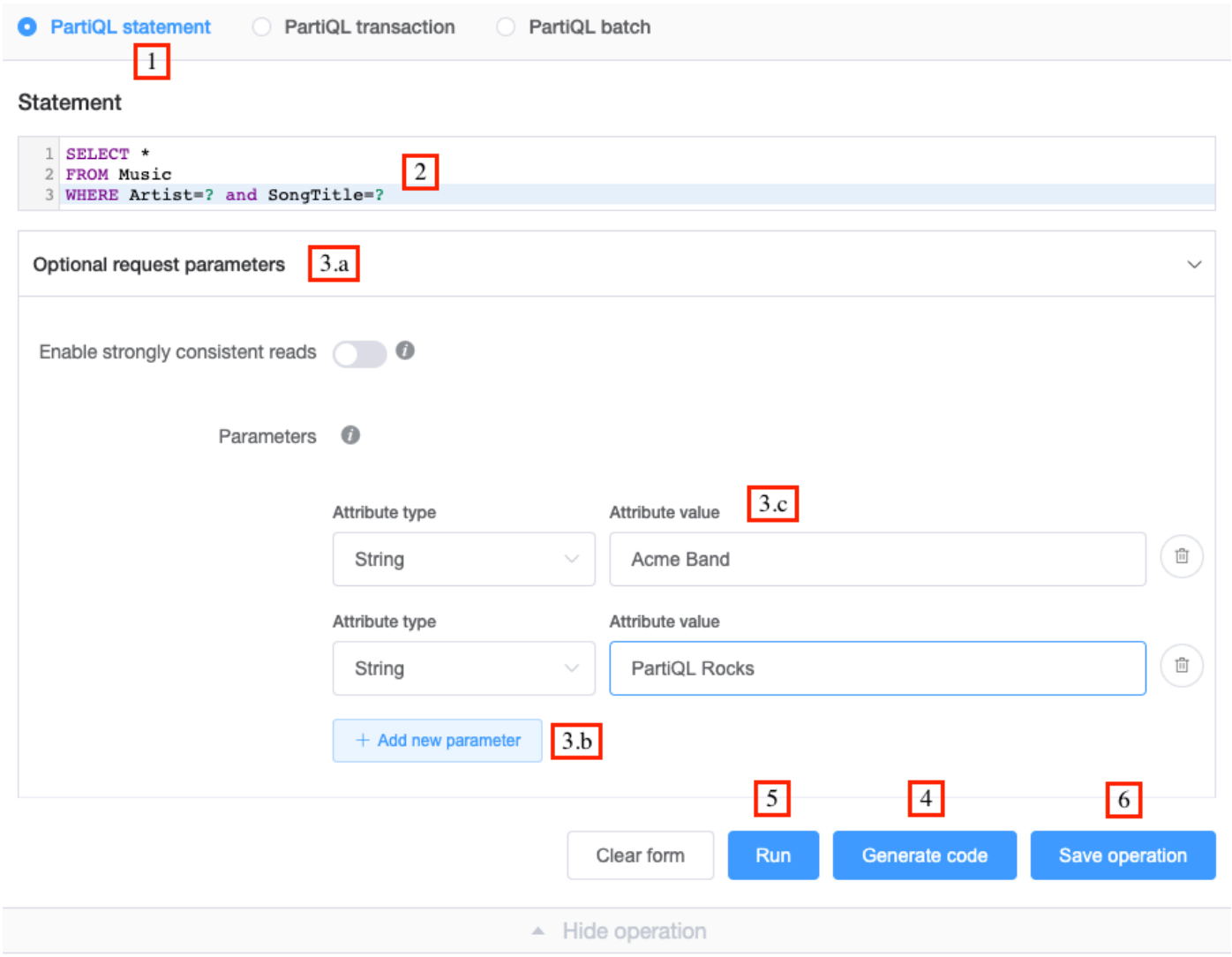
Console



1. Sign in to the AWS Management Console and open the DynamoDB console at https://console.aws.amazon.com/dynamodb/.

2. In the navigation pane on the left side of the console, choose **PartiQL editor**.
3. Choose the **Music** table.
4. Choose **Query table**. This action generates a query that will not result in a full table scan.
5. Replace `partitionKeyValue` with the string value `Acme Band`. Replace `sortKeyValue` with the string value `Happy Day`.
6. Choose the **Run** button.
7. You can view the results of the query by choosing the **Table view** or the **JSON view** buttons.

NoSQL workbench



1. Choose **PartiQL statement**.

2. Enter the following PartiQL SELECT statement

```
SELECT *
FROM Music
WHERE Artist=? and SongTitle=?
```

3. To specify a value for the `Artist` and `SongTitle` parameters:

   a. Choose **Optional request parameters**.

   b. Choose **Add new parameters**.

   c. Choose the attribute type **string** and value `Acme Band`.

   d. Repeat steps b and c, and choose type **string** and value `PartiQL Rocks`.

4. If you want to generate code, choose **Generate code**.

   Select your desired language from the displayed tabs. You can now copy this code and use it in your application.

5. If you want the operation to be run immediately, choose **Run**.

AWS CLI

1. Create an item in the `Music` table using the INSERT PartiQL statement.

```
aws dynamodb execute-statement --statement "INSERT INTO Music  \
         VALUE  \
         {'Artist':'Acme Band','SongTitle':'PartiQL Rocks'}"
```

2. Retrieve an item from the Music table using the SELECT PartiQL statement.

```
aws dynamodb execute-statement --statement "SELECT * FROM Music   \
                                            WHERE Artist='Acme Band' AND
 SongTitle='PartiQL Rocks'"
```

3. Update an item in the `Music` table using the UPDATE PartiQL statement.

```
aws dynamodb execute-statement --statement "UPDATE Music  \
                                            SET AwardsWon=1  \
                                            SET AwardDetail={'Grammys':[2020,
 2018]}  \
```

```
                                            WHERE Artist='Acme Band' AND
 SongTitle='PartiQL Rocks'"
```

Add a list value for an item in the `Music` table.

```
aws dynamodb execute-statement --statement "UPDATE Music  \
                                            SET AwardDetail.Grammys
 =list_append(AwardDetail.Grammys,[2016])  \
                                            WHERE Artist='Acme Band' AND
 SongTitle='PartiQL Rocks'"
```

Remove a list value for an item in the `Music` table.

```
aws dynamodb execute-statement --statement "UPDATE Music  \
                                            REMOVE AwardDetail.Grammys[2]  \
                                            WHERE Artist='Acme Band' AND
 SongTitle='PartiQL Rocks'"
```

Add a new map member for an item in the `Music` table.

```
aws dynamodb execute-statement --statement "UPDATE Music  \
                                            SET AwardDetail.BillBoard=[2020]  \
                                            WHERE Artist='Acme Band' AND
 SongTitle='PartiQL Rocks'"
```

Add a new string set attribute for an item in the `Music` table.

```
aws dynamodb execute-statement --statement "UPDATE Music  \
                                            SET BandMembers =<<'member1',
 'member2'>>  \
                                            WHERE Artist='Acme Band' AND
 SongTitle='PartiQL Rocks'"
```

Update a string set attribute for an item in the `Music` table.

```
aws dynamodb execute-statement --statement "UPDATE Music  \
                                            SET BandMembers
 =set_add(BandMembers, <<'newmember'>>)  \
```

```
                                                WHERE Artist='Acme Band' AND
 SongTitle='PartiQL Rocks'"
```

4. Delete an item from the `Music` table using the DELETE PartiQL statement.

```
aws dynamodb execute-statement --statement "DELETE  FROM Music  \
    WHERE Artist='Acme Band' AND SongTitle='PartiQL Rocks'"
```

Java

```
import java.util.ArrayList;
import java.util.List;

import com.amazonaws.AmazonClientException;
import com.amazonaws.AmazonServiceException;
import software.amazon.dynamodb.AmazonDynamoDB;
import software.amazon.dynamodb.AmazonDynamoDBClientBuilder;
import software.amazon.dynamodb.model.AttributeValue;
import software.amazon.dynamodb.model.ConditionalCheckFailedException;
import software.amazon.dynamodb.model.ExecuteStatementRequest;
import software.amazon.dynamodb.model.ExecuteStatementResult;
import software.amazon.dynamodb.model.InternalServerErrorException;
import software.amazon.dynamodb.model.ItemCollectionSizeLimitExceededException;
import software.amazon.dynamodb.model.ProvisionedThroughputExceededException;
import software.amazon.dynamodb.model.RequestLimitExceededException;
import software.amazon.dynamodb.model.ResourceNotFoundException;
import software.amazon.dynamodb.model.TransactionConflictException;

public class DynamoDBPartiQGettingStarted {

    public static void main(String[] args) {
        // Create the DynamoDB Client with the region you want
        AmazonDynamoDB dynamoDB = createDynamoDbClient("us-west-1");

        try {
            // Create ExecuteStatementRequest
            ExecuteStatementRequest executeStatementRequest = new
 ExecuteStatementRequest();
            List<AttributeValue> parameters= getPartiQLParameters();

            //Create an item in the Music table using the INSERT PartiQL statement
```

```
            processResults(executeStatementRequest(dynamoDB, "INSERT INTO Music
value {'Artist':?,'SongTitle':?}", parameters));

            //Retrieve an item from the Music table using the SELECT PartiQL
statement.
            processResults(executeStatementRequest(dynamoDB, "SELECT * FROM Music
where Artist=? and SongTitle=?", parameters));

            //Update an item in the Music table using the UPDATE PartiQL statement.
            processResults(executeStatementRequest(dynamoDB, "UPDATE Music
SET AwardsWon=1 SET AwardDetail={'Grammys':[2020, 2018]}  where Artist=? and
SongTitle=?", parameters));

            //Add a list value for an item in the Music table.
            processResults(executeStatementRequest(dynamoDB, "UPDATE Music SET
AwardDetail.Grammys =list_append(AwardDetail.Grammys,[2016])  where Artist=? and
SongTitle=?", parameters));

            //Remove a list value for an item in the Music table.
            processResults(executeStatementRequest(dynamoDB, "UPDATE Music REMOVE
AwardDetail.Grammys[2]   where Artist=? and SongTitle=?", parameters));

            //Add a new map member for an item in the Music table.
            processResults(executeStatementRequest(dynamoDB, "UPDATE Music set
AwardDetail.BillBoard=[2020] where Artist=? and SongTitle=?", parameters));

            //Add a new string set attribute for an item in the Music table.
            processResults(executeStatementRequest(dynamoDB, "UPDATE Music
SET BandMembers =<<'member1', 'member2'>> where Artist=? and SongTitle=?",
parameters));

            //update a string set attribute for an item in the Music table.
            processResults(executeStatementRequest(dynamoDB, "UPDATE Music SET
BandMembers =set_add(BandMembers, <<'newmember'>>) where Artist=? and SongTitle=?",
parameters));

            //Retrieve an item from the Music table using the SELECT PartiQL
statement.
            processResults(executeStatementRequest(dynamoDB, "SELECT * FROM Music
where Artist=? and SongTitle=?", parameters));

            //delete an item from the Music Table
            processResults(executeStatementRequest(dynamoDB, "DELETE  FROM Music
where Artist=? and SongTitle=?", parameters));
```

```
        } catch (Exception e) {
            handleExecuteStatementErrors(e);
        }
    }

    private static AmazonDynamoDB createDynamoDbClient(String region) {
        return AmazonDynamoDBClientBuilder.standard().withRegion(region).build();
    }

    private static List<AttributeValue> getPartiQLParameters() {
        List<AttributeValue> parameters = new ArrayList<AttributeValue>();
        parameters.add(new AttributeValue("Acme Band"));
        parameters.add(new AttributeValue("PartiQL Rocks"));
        return parameters;
    }

    private static ExecuteStatementResult executeStatementRequest(AmazonDynamoDB
client, String statement, List<AttributeValue> parameters ) {
        ExecuteStatementRequest request = new ExecuteStatementRequest();
        request.setStatement(statement);
        request.setParameters(parameters);
        return client.executeStatement(request);
    }

    private static void processResults(ExecuteStatementResult
executeStatementResult) {
        System.out.println("ExecuteStatement successful: "+
executeStatementResult.toString());

    }

    // Handles errors during ExecuteStatement execution. Use recommendations in
error messages below to add error handling specific to
    // your application use-case.
    private static void handleExecuteStatementErrors(Exception exception) {
        try {
            throw exception;
        } catch (ConditionalCheckFailedException ccfe) {
            System.out.println("Condition check specified in the operation failed,
review and update the condition " +
                                       "check before retrying. Error: " +
ccfe.getErrorMessage());
        } catch (TransactionConflictException tce) {
```

```
            System.out.println("Operation was rejected because there is an ongoing
transaction for the item, generally " +
                                       "safe to retry with exponential back-off.
Error: " + tce.getErrorMessage());
        } catch (ItemCollectionSizeLimitExceededException icslee) {
            System.out.println("An item collection is too large, you\'re using Local
Secondary Index and exceeded " +
                                       "size limit of items per
partition key. Consider using Global Secondary Index instead. Error: " +
icslee.getErrorMessage());
        } catch (Exception e) {
            handleCommonErrors(e);
        }
    }

    private static void handleCommonErrors(Exception exception) {
        try {
            throw exception;
        } catch (InternalServerErrorException isee) {
            System.out.println("Internal Server Error, generally safe to retry with
exponential back-off. Error: " + isee.getErrorMessage());
        } catch (RequestLimitExceededException rlee) {
            System.out.println("Throughput exceeds the current throughput limit for
your account, increase account level throughput before " +
                                       "retrying. Error: " +
rlee.getErrorMessage());
        } catch (ProvisionedThroughputExceededException ptee) {
            System.out.println("Request rate is too high. If you're using a custom
retry strategy make sure to retry with exponential back-off. " +
                                       "Otherwise consider reducing frequency of
requests or increasing provisioned capacity for your table or secondary index.
Error: " +
                                       ptee.getErrorMessage());
        } catch (ResourceNotFoundException rnfe) {
            System.out.println("One of the tables was not found, verify table exists
before retrying. Error: " + rnfe.getErrorMessage());
        } catch (AmazonServiceException ase) {
            System.out.println("An AmazonServiceException occurred, indicates that
the request was correctly transmitted to the DynamoDB " +
                                       "service, but for some reason, the service
was not able to process it, and returned an error response instead. Investigate and
" +
                                       "configure retry strategy. Error type: " +
ase.getErrorType() + ". Error message: " + ase.getErrorMessage());
```

```
        } catch (AmazonClientException ace) {
            System.out.println("An AmazonClientException occurred, indicates that
 the client was unable to get a response from DynamoDB " +
                                       "service, or the client was unable to parse
 the response from the service. Investigate and configure retry strategy. "+
                                       "Error: " + ace.getMessage());
        } catch (Exception e) {
            System.out.println("An exception occurred, investigate and configure
 retry strategy. Error: " + e.getMessage());
        }
    }

}
```

Using parameterized statements

Instead of embedding values directly in a PartiQL statement string, you can use question mark (?) placeholders and supply the values separately in the `Parameters` field. Each ? is replaced by the corresponding parameter value, in the order they are provided.

Using parameterized statements is a best practice because it separates the statement structure from the data values, making statements easier to read and reuse. It also avoids the need to manually format and escape attribute values in the statement string.

Parameterized statements are supported in `ExecuteStatement`, `BatchExecuteStatement`, and `ExecuteTransaction` operations.

The following examples retrieve an item from the `Music` table using parameterized values for the partition key and sort key.

AWS CLI parameterized

```
aws dynamodb execute-statement \
    --statement "SELECT * FROM \"Music\" WHERE Artist=? AND SongTitle=?" \
    --parameters '[{"S": "Acme Band"}, {"S": "PartiQL Rocks"}]'
```

Java parameterized

```
List<AttributeValue> parameters = new ArrayList<>();
parameters.add(new AttributeValue("Acme Band"));
parameters.add(new AttributeValue("PartiQL Rocks"));
```

```
ExecuteStatementRequest request = new ExecuteStatementRequest()
    .withStatement("SELECT * FROM Music WHERE Artist=? AND SongTitle=?")
    .withParameters(parameters);

ExecuteStatementResult result = dynamoDB.executeStatement(request);
```

Python parameterized

```
response = dynamodb_client.execute_statement(
    Statement="SELECT * FROM Music WHERE Artist=? AND SongTitle=?",
    Parameters=[
        {'S': 'Acme Band'},
        {'S': 'PartiQL Rocks'}
    ]
)
```

Note

The Java example in the preceding getting started section uses parameterized statements throughout. The `getPartiQLParameters()` method builds the parameter list, and each statement uses ? placeholders instead of inline values.

PartiQL data types for DynamoDB

The following table lists the data types you can use with PartiQL for DynamoDB.

DynamoDB data type	PartiQL representation	Notes
`Boolean`	TRUE | FALSE	Not case sensitive.
`Binary`	N/A	Only supported via code.
`List`	[value1, value2,...]	There are no restrictions on the data types that can be stored in a List type, and the elements in a List do not have to be of the same type.

DynamoDB data type	PartiQL representation	Notes
`Map`	{ 'name' : value }	There are no restrictions on the data types that can be stored in a Map type, and the elements in a Map do not have to be of the same type.
`Null`	NULL	Not case sensitive.
`Number`	1, 1.0, 1e0	Numbers can be positive, negative, or zero. Numbers can have up to 38 digits of precision.
`Number Set`	<<number1, number2>>	The elements in a number set must be of type Number.
`String Set`	<<'string1', 'string2'>>	The elements in a string set must be of type String.
`String`	'string value'	Single quotes must be used to specify String values.

Examples

The following statement demonstrates how to insert the following data types: `String`, `Number`, `Map`, `List`, `Number Set` and `String Set`.

```
INSERT INTO TypesTable value {'primarykey':'1',
'NumberType':1,
'MapType' : {'entryname1': 'value', 'entryname2': 4},
'ListType': [1,'stringval'],
'NumberSetType':<<1,34,32,4.5>>,
'StringSetType':<<'stringval','stringval2'>>
}
```

The following statement demonstrates how to insert new elements into the `Map`, `List`, `Number Set` and `String Set` types and change the value of a `Number` type.

```
UPDATE TypesTable
SET NumberType=NumberType + 100
SET MapType.NewMapEntry=[2020, 'stringvalue', 2.4]
SET ListType = LIST_APPEND(ListType, [4, <<'string1', 'string2'>>])
SET NumberSetType= SET_ADD(NumberSetType, <<345, 48.4>>)
SET StringSetType = SET_ADD(StringSetType, <<'stringsetvalue1', 'stringsetvalue2'>>)
WHERE primarykey='1'
```

The following statement demonstrates how to remove elements from the `Map`, `List`, `Number Set` and `String Set` types and change the value of a `Number` type.

```
UPDATE TypesTable
SET NumberType=NumberType - 1
REMOVE ListType[1]
REMOVE MapType.NewMapEntry
SET NumberSetType = SET_DELETE( NumberSetType, <<345>>)
SET StringSetType = SET_DELETE( StringSetType, <<'stringsetvalue1'>>)
WHERE primarykey='1'
```

For more information, see DynamoDB data types.

PartiQL statements for DynamoDB

Amazon DynamoDB supports the following PartiQL statements.

> **Note**
>
> DynamoDB does not support all PartiQL statements.
> This reference provides basic syntax and usage examples of PartiQL statements that you manually run using the AWS CLI or APIs.

Data manipulation language (DML) is the set of PartiQL statements that you use to manage data in DynamoDB tables. You use DML statements to add, modify, or delete data in a table.

The following DML and query language statements are supported:

- PartiQL select statements for DynamoDB
- PartiQL update statements for DynamoDB

- PartiQL insert statements for DynamoDB
- PartiQL delete statements for DynamoDB

Performing transactions with PartiQL for DynamoDB and Running batch operations with PartiQL for DynamoDB are also supported by PartiQL for DynamoDB.

PartiQL select statements for DynamoDB

Use the SELECT statement to retrieve data from a table in Amazon DynamoDB.

Using the SELECT statement can result in a full table scan if an equality or IN condition with a partition key is not provided in the WHERE clause. A scan operation examines every item for the requested values and can use up the provisioned throughput for a large table or index in a single operation.

If you want to avoid full table scan in PartiQL, you can:

- Author your SELECT statements to not result in full table scans by making sure your WHERE clause condition is configured accordingly.
- Disable full table scans using the IAM policy specified at Example: Allow select statements and deny full table scan statements in PartiQL for DynamoDB, in the DynamoDB developer guide.

For more information see Best practices for querying and scanning data, in the DynamoDB developer guide.

Topics

- Syntax
- Parameters
- Examples

Syntax

```
SELECT expression  [, ...]
FROM table[.index]
[ WHERE condition ] [ [ORDER BY key [DESC|ASC] , ...]
```

Parameters

expression

(Required) A projection formed from the * wildcard or a projection list of one or more attribute names or document paths from the result set. An expression can consist of calls to Use PartiQL functions with DynamoDB or fields that are modified by PartiQL arithmetic, comparison, and logical operators for DynamoDB.

table

(Required) The table name to query.

index

(Optional) The name of the index to query.

> **Note**
>
> You must add double quotation marks to the table name and index name when querying an index.
>
> ```
> SELECT *
> FROM "TableName"."IndexName"
> ```

condition

(Optional) The selection criteria for the query.

> **Important**
>
> To ensure that a `SELECT` statement does not result in a full table scan, the `WHERE` clause condition must specify a partition key. Use the equality or IN operator.
> For example, if you have an `Orders` table with an `OrderID` partition key and other non-key attributes, including an `Address`, the following statements would not result in a full table scan:
>
> ```
> SELECT *
> FROM "Orders"
> ```

```
WHERE OrderID = 100

SELECT *
FROM "Orders"
WHERE OrderID = 100 and Address='some address'

SELECT *
FROM "Orders"
WHERE OrderID = 100 or OrderID = 200

SELECT *
FROM "Orders"
WHERE OrderID IN [100, 300, 234]
```

The following SELECT statements, however, will result in a full table scan:

```
SELECT *
FROM "Orders"
WHERE OrderID > 1

SELECT *
FROM "Orders"
WHERE Address='some address'

SELECT *
FROM "Orders"
WHERE OrderID = 100 OR Address='some address'
```

key

(Optional) A hash key or a sort key to use to order returned results. The default order is ascending (ASC) specify DESC if you want the results retuned in descending order.

Note

If you omit the WHERE clause, then all of the items in the table are retrieved.

Examples

The following query returns one item, if one exists, from the `Orders` table by specifying the partition key, `OrderID`, and using the equality operator.

```
SELECT OrderID, Total
FROM "Orders"
WHERE OrderID = 1
```

The following query returns all items in the `Orders` table that have a specific partition key, `OrderID`, values using the OR operator.

```
SELECT OrderID, Total
FROM "Orders"
WHERE OrderID = 1 OR OrderID = 2
```

The following query returns all items in the `Orders` table that have a specific partition key, `OrderID`, values using the IN operator. The returned results are in descending order, based on the `OrderID` key attribute value.

```
SELECT OrderID, Total
FROM "Orders"
WHERE OrderID IN [1, 2, 3] ORDER BY OrderID DESC
```

The following query shows a full table scan that returns all items from the `Orders` table that have a `Total` greater than 500, where `Total` is a non-key attribute.

```
SELECT OrderID, Total
FROM "Orders"
WHERE Total > 500
```

The following query shows a full table scan that returns all items from the `Orders` table within a specific `Total` order range, using the IN operator and a non-key attribute `Total`.

```
SELECT OrderID, Total
FROM "Orders"
WHERE Total IN [500, 600]
```

The following query shows a full table scan that returns all items from the `Orders` table within a specific `Total` order range, using the BETWEEN operator and a non-key attribute `Total`.

```
SELECT OrderID, Total
FROM "Orders"
WHERE Total BETWEEN 500 AND 600
```

The following query returns the first date a firestick device was used to watch by specifying the partition key `CustomerID` and sort key `MovieID` in the WHERE clause condition and using document paths in the SELECT clause.

```
SELECT Devices.FireStick.DateWatched[0]
FROM WatchList
WHERE CustomerID= 'C1' AND MovieID= 'M1'
```

The following query shows a full table scan that returns the list of items where a firestick device was first used after 12/24/19 using document paths in the WHERE clause condition.

```
SELECT Devices
FROM WatchList
WHERE Devices.FireStick.DateWatched[0] >= '12/24/19'
```

PartiQL update statements for DynamoDB

Use the `UPDATE` statement to modify the value of one or more attributes within an item in an Amazon DynamoDB table.

> **Note**
>
> You can only update one item at a time; you cannot issue a single DynamoDB PartiQL statement that updates multiple items. For information on updating multiple items, see Performing transactions with PartiQL for DynamoDB or Running batch operations with PartiQL for DynamoDB.

Topics

- Syntax
- Parameters
- Return value
- Examples

Syntax

```
UPDATE  table
[SET | REMOVE]  path  [=  data] […]
WHERE condition [RETURNING returnvalues]
<returnvalues>  ::= [ALL OLD | MODIFIED OLD | ALL NEW | MODIFIED NEW] *
```

Parameters

table

(Required) The table containing the data to be modified.

path

(Required) An attribute name or document path to be created or modified.

data

(Required) An attribute value or the result of an operation.

The supported operations to use with SET:

- LIST_APPEND: adds a value to a list type.
- SET_ADD: adds a value to a number or string set.
- SET_DELETE: removes a value from a number or string set.

condition

(Required) The selection criteria for the item to be modified. This condition must resolve to a single primary key value.

returnvalues

(Optional) Use `returnvalues` if you want to get the item attributes as they appear before or after they are updated. The valid values are:

- `ALL OLD *`- Returns all of the attributes of the item, as they appeared before the update operation.
- `MODIFIED OLD *`- Returns only the updated attributes, as they appeared before the update operation.
- `ALL NEW *`- Returns all of the attributes of the item, as they appear after the update operation.

- `MODIFIED NEW *`- Returns only the updated attributes, as they appear after the `UpdateItem` operation.

Return value

This statement does not return a value unless `returnvalues` parameter is specified.

> **Note**
>
> If the WHERE clause of the UPDATE statement does not evaluate to true for any item in the DynamoDB table, `ConditionalCheckFailedException` is returned.

Examples

Update an attribute value in an existing item. If the attribute does not exist, it is created.

The following query updates an item in the `"Music"` table by adding an attribute of type number (`AwardsWon`) and an attribute of type map (`AwardDetail`).

```
UPDATE "Music"
SET AwardsWon=1
SET AwardDetail={'Grammys':[2020, 2018]}
WHERE Artist='Acme Band' AND SongTitle='PartiQL Rocks'
```

You can add `RETURNING ALL OLD *` to return the attributes as they appeared before the `Update` operation.

```
UPDATE "Music"
SET AwardsWon=1
SET AwardDetail={'Grammys':[2020, 2018]}
WHERE Artist='Acme Band' AND SongTitle='PartiQL Rocks'
RETURNING ALL OLD *
```

This returns the following:

```
{
    "Items": [
        {
```

```
            "Artist": {
                "S": "Acme Band"
            },
            "SongTitle": {
                "S": "PartiQL Rocks"
            }
        }
    ]
}
```

You can add `RETURNING ALL NEW *` to return the attributes as they appeared after the `Update` operation.

```
UPDATE "Music"
SET AwardsWon=1
SET AwardDetail={'Grammys':[2020, 2018]}
WHERE Artist='Acme Band' AND SongTitle='PartiQL Rocks'
RETURNING ALL NEW *
```

This returns the following:

```
{
    "Items": [
        {
            "AwardDetail": {
                "M": {
                    "Grammys": {
                        "L": [
                            {
                                "N": "2020"
                            },
                            {
                                "N": "2018"
                            }
                        ]
                    }
                }
            },
            "AwardsWon": {
                "N": "1"
            }
        }
    ]
```

```
}
```

The following query updates an item in the "Music" table by appending to a list AwardDetail.Grammys.

```
UPDATE "Music"
SET AwardDetail.Grammys =list_append(AwardDetail.Grammys,[2016])
WHERE Artist='Acme Band' AND SongTitle='PartiQL Rocks'
```

The following query updates an item in the "Music" table by removing from a list AwardDetail.Grammys.

```
UPDATE "Music"
REMOVE AwardDetail.Grammys[2]
WHERE Artist='Acme Band' AND SongTitle='PartiQL Rocks'
```

The following query updates an item in the "Music" table by adding BillBoard to the map AwardDetail.

```
UPDATE "Music"
SET AwardDetail.BillBoard=[2020]
WHERE Artist='Acme Band' AND SongTitle='PartiQL Rocks'
```

The following query updates an item in the "Music" table by adding the string set attribute BandMembers.

```
UPDATE "Music"
SET BandMembers =<<'member1', 'member2'>>
WHERE Artist='Acme Band' AND SongTitle='PartiQL Rocks'
```

The following query updates an item in the "Music" table by adding newbandmember to the string set BandMembers.

```
UPDATE "Music"
SET BandMembers =set_add(BandMembers, <<'newbandmember'>>)
WHERE Artist='Acme Band' AND SongTitle='PartiQL Rocks'
```

PartiQL delete statements for DynamoDB

Use the DELETE statement to delete an existing item from your Amazon DynamoDB table.

> **Note**
>
> You can only delete one item at a time. You cannot issue a single DynamoDB PartiQL statement that deletes multiple items. For information on deleting multiple items, see Performing transactions with PartiQL for DynamoDB or Running batch operations with PartiQL for DynamoDB.

Topics

- Syntax
- Parameters
- Return value
- Examples

Syntax

```
DELETE FROM table
 WHERE condition [RETURNING returnvalues]
 <returnvalues>  ::= ALL OLD *
```

Parameters

table

(Required) The DynamoDB table containing the item to be deleted.

condition

(Required) The selection criteria for the item to be deleted; this condition must resolve to a single primary key value.

returnvalues

(Optional) Use `returnvalues` if you want to get the item attributes as they appeared before they were deleted. The valid values are:

- `ALL OLD *`- The content of the old item is returned.

Return value

This statement does not return a value unless `returnvalues` parameter is specified.

> **Note**
>
> If the DynamoDB table does not have any item with the same primary key as that of the item for which the DELETE is issued, SUCCESS is returned with 0 items deleted. If the table has an item with same primary key, but the condition in the WHERE clause of the DELETE statement evaluates to false, `ConditionalCheckFailedException` is returned.

Examples

The following query deletes an item in the `"Music"` table.

```
DELETE FROM "Music" WHERE "Artist" = 'Acme Band' AND "SongTitle" = 'PartiQL Rocks'
```

You can add the parameter `RETURNING ALL OLD *` to return the data that was deleted.

```
DELETE FROM "Music" WHERE "Artist" = 'Acme Band' AND "SongTitle" = 'PartiQL Rocks'
 RETURNING ALL OLD *
```

The `Delete` statement now returns the following:

```
{
    "Items": [
        {
            "Artist": {
                "S": "Acme Band"
            },
            "SongTitle": {
                "S": "PartiQL Rocks"
            }
        }
    ]
}
```

PartiQL insert statements for DynamoDB

Use the `INSERT` statement to add an item to a table in Amazon DynamoDB.

> **Note**
>
> You can only insert one item at a time; you cannot issue a single DynamoDB PartiQL statement that inserts multiple items. For information on inserting multiple items, see Performing transactions with PartiQL for DynamoDB or Running batch operations with PartiQL for DynamoDB.

Topics

- Syntax
- Parameters
- Return value
- Examples

Syntax

Insert a single item.

```
INSERT INTO table VALUE item
```

Parameters

table

(Required) The table where you want to insert the data. The table must already exist.

item

(Required) A valid DynamoDB item represented as a PartiQL tuple. You must specify only *one* item and each attribute name in the item is case-sensitive and can be denoted with *single* quotation marks (`'...'`) in PartiQL.

String values are also denoted with *single* quotation marks (`'...'`) in PartiQL.

Return value

This statement does not return any values.

> **Note**
>
> If the DynamoDB table already has an item with the same primary key as the primary key of the item being inserted, `DuplicateItemException` is returned.

Examples

```
INSERT INTO "Music" value {'Artist' : 'Acme Band','SongTitle' : 'PartiQL Rocks'}
```

Use PartiQL functions with DynamoDB

PartiQL in Amazon DynamoDB supports the following built-in variants of SQL standard functions.

> **Note**
>
> Any SQL functions that are not included in this list are not currently supported in DynamoDB.

Aggregate functions

- Using the SIZE function with PartiQL for amazon DynamoDB

Conditional functions

- Using the EXISTS function with PartiQL for DynamoDB
- Using the ATTRIBUTE_TYPE function with PartiQL for DynamoDB
- Using the BEGINS_WITH function with PartiQL for DynamoDB
- Using the CONTAINS function with PartiQL for DynamoDB
- Using the MISSING function with PartiQL for DynamoDB

Using the EXISTS function with PartiQL for DynamoDB

You can use EXISTS to perform the same function as `ConditionCheck` does in the TransactWriteItems API. The EXISTS function can only be used in transactions.

Given a value, returns TRUE if the value is a non-empty collection. Otherwise, returns FALSE.

> **Note**
>
> This function can only be used in transactional operations.

Syntax

```
EXISTS ( statement )
```

Arguments

statement

(Required) The SELECT statement that the function evaluates.

> **Note**
>
> The SELECT statement must specify a full primary key and one other condition.

Return type

bool

Examples

```
EXISTS(
    SELECT * FROM "Music"
    WHERE "Artist" = 'Acme Band' AND "SongTitle" = 'PartiQL Rocks')
```

Using the BEGINS_WITH function with PartiQL for DynamoDB

Returns TRUE if the attribute specified begins with a particular substring.

Syntax

```
begins_with(path, value )
```

Arguments

path

(Required) The attribute name or document path to use.

value

(Required) The string to search for.

Return type

bool

Examples

```
SELECT * FROM "Orders" WHERE "OrderID"=1 AND begins_with("Address", '7834 24th')
```

Using the MISSING function with PartiQL for DynamoDB

Returns TRUE if the item does not contain the attribute specified. Only equality and inequality operators can be used with this function.

Syntax

```
 attributename IS | IS NOT  MISSING
```

Arguments

attributename

(Required) The attribute name to look for.

Return type

bool

Examples

```
SELECT * FROM Music WHERE "Awards" is MISSING
```

Using the ATTRIBUTE_TYPE function with PartiQL for DynamoDB

Returns TRUE if the attribute at the specified path is of a particular data type.

Syntax

```
attribute_type( attributename, type )
```

Arguments

attributename

(Required) The attribute name to use.

type

(Required) The attribute type to check for. For a list of valid values, see DynamoDB attribute_type.

Return type

`bool`

Examples

```
SELECT * FROM "Music" WHERE attribute_type("Artist", 'S')
```

Using the CONTAINS function with PartiQL for DynamoDB

Returns TRUE if the attribute specified by the path is one of the following:

- A String that contains a particular substring.
- A Set that contains a particular element within the set.

For more information, see the DynamoDB contains function.

Syntax

```
contains( path, substring )
```

Arguments

path

(Required) The attribute name or document path to use.

substring

(Required) The attribute substring or set member to check for. For more information, see the DynamoDB contains function.

Return type

`bool`

Examples

```
SELECT * FROM "Orders" WHERE "OrderID"=1 AND contains("Address", 'Kirkland')
```

Using the SIZE function with PartiQL for amazon DynamoDB

Returns a number representing an attribute's size in bytes. The following are valid data types for use with size. For more information, see the DynamoDB size function.

Syntax

```
size( path)
```

Arguments

path

(Required) The attribute name or document path.

For supported types, see DynamoDB size function.

Return type

`int`

Examples

```
SELECT * FROM "Orders" WHERE "OrderID"=1 AND size("Image") >300
```

PartiQL arithmetic, comparison, and logical operators for DynamoDB

PartiQL in Amazon DynamoDB supports the following SQL standard operators.

> **Note**
>
> Any SQL operators that are not included in this list are not currently supported in DynamoDB.

Arithmetic operators

Operator	Description
+	Add
-	Subtract

Comparison operators

Operator	Description
=	Equal to
<>	Not Equal to
!=	Not Equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

Logical operators

Operator	Description
AND	TRUE if all the conditions separated by AND are TRUE
BETWEEN	TRUE if the operand is within the range of comparisons. This operator is inclusive of the lower and upper bound of the operands on which you apply it.
IN	TRUE if the operand is equal to one of a list of expressions (at max 50 hash attribute values or at max 100 non-key attribute values). Results are returned in pages of up to 10 items. If the IN list contains more values, you must use the NextToken returned in the response to retrieve subsequent pages.
IS	TRUE if the operand is a given, PartiQL data type, including NULL or MISSING
NOT	Reverses the value of a given Boolean expression
OR	TRUE if any of the conditions separated by OR are TRUE

For more information about using logical operators, see Making comparisons and Logical evaluations.

Performing transactions with PartiQL for DynamoDB

This section describes how to use transactions with PartiQL for DynamoDB. PartiQL transactions are limited to 100 total statements (actions).

For more information on DynamoDB transactions, see Managing complex workflows with DynamoDB transactions.

> **Note**
>
> The entire transaction must consist of either read statements or write statements. You can't mix both in one transaction. The EXISTS function is an exception. You can use it to check the condition of specific attributes of the item in a similar manner to `ConditionCheck` in the TransactWriteItems API operation.

Topics

- Syntax
- Parameters
- Return values
- Examples

Syntax

```
[
   {
      "Statement":" statement ",
      "Parameters":[
         {
            " parametertype " : " parametervalue "
         }, ...]
   } , ...
]
```

Parameters

statement

(Required) A PartiQL for DynamoDB supported statement.

> **Note**
>
> The entire transaction must consist of either read statements or write statements. You can't mix both in one transaction.

parametertype

(Optional) A DynamoDB type, if parameters were used when specifying the PartiQL statement.

parametervalue

(Optional) A parameter value if parameters were used when specifying the PartiQL statement.

Return values

This statement doesn't return any values for Write operations (INSERT, UPDATE, or DELETE). However, it returns different values for Read operations (SELECT) based on the conditions specified in the WHERE clause.

> **Note**
>
> If any of the singleton INSERT, UPDATE, or DELETE operations return an error, the transactions are canceled with the `TransactionCanceledException` exception, and the cancellation reason code includes the errors from the individual singleton operations.

Examples

The following example runs multiple statements as a transaction.

AWS CLI

1. Save the following JSON code to a file called partiql.json.

```
[
    {
        "Statement": "EXISTS(SELECT * FROM \"Music\" where Artist='No One You
 Know' and SongTitle='Call Me Today' and Awards is  MISSING)"
    },
    {
```

```
        "Statement": "INSERT INTO Music value {'Artist':?,'SongTitle':'?'}",
        "Parameters": [{\"S\": \"Acme Band\"}, {\"S\": \"Best Song\"}]
    },
    {
        "Statement": "UPDATE \"Music\" SET AwardsWon=1 SET
 AwardDetail={'Grammys':[2020, 2018]}  where Artist='Acme Band' and
 SongTitle='PartiQL Rocks'"
    }
]
```

2. Run the following command in a command prompt.

```
aws dynamodb execute-transaction --transact-statements  file://partiql.json
```

Java

```
public class DynamoDBPartiqlTransaction {

    public static void main(String[] args) {
        // Create the DynamoDB Client with the region you want
        AmazonDynamoDB dynamoDB = createDynamoDbClient("us-west-2");

        try {
            // Create ExecuteTransactionRequest
            ExecuteTransactionRequest executeTransactionRequest =
 createExecuteTransactionRequest();
            ExecuteTransactionResult executeTransactionResult =
 dynamoDB.executeTransaction(executeTransactionRequest);
            System.out.println("ExecuteTransaction successful.");
            // Handle executeTransactionResult

        } catch (Exception e) {
            handleExecuteTransactionErrors(e);
        }
    }

    private static AmazonDynamoDB createDynamoDbClient(String region) {
        return AmazonDynamoDBClientBuilder.standard().withRegion(region).build();
    }

    private static ExecuteTransactionRequest createExecuteTransactionRequest() {
        ExecuteTransactionRequest request = new ExecuteTransactionRequest();
```

```
        // Create statements
        List<ParameterizedStatement> statements = getPartiQLTransactionStatements();

        request.setTransactStatements(statements);
        return request;
    }

    private static List<ParameterizedStatement> getPartiQLTransactionStatements() {
        List<ParameterizedStatement> statements = new
ArrayList<ParameterizedStatement>();

        statements.add(new ParameterizedStatement()
                                .withStatement("EXISTS(SELECT * FROM "Music" where
Artist='No One You Know' and SongTitle='Call Me Today' and Awards is  MISSING)"));

        statements.add(new ParameterizedStatement()
                                .withStatement("INSERT INTO "Music" value
{'Artist':'?','SongTitle':'?'}")
                                .withParameters(new AttributeValue("Acme Band"),new
AttributeValue("Best Song")));

        statements.add(new ParameterizedStatement()
                                .withStatement("UPDATE "Music" SET AwardsWon=1
SET AwardDetail={'Grammys':[2020, 2018]}  where Artist='Acme Band' and
SongTitle='PartiQL Rocks'"));

        return statements;
    }

    // Handles errors during ExecuteTransaction execution. Use recommendations in
error messages below to add error handling specific to
    // your application use-case.
    private static void handleExecuteTransactionErrors(Exception exception) {
        try {
            throw exception;
        } catch (TransactionCanceledException tce) {
            System.out.println("Transaction Cancelled, implies a client issue, fix
before retrying. Error: " + tce.getErrorMessage());
        } catch (TransactionInProgressException tipe) {
            System.out.println("The transaction with the given request token is
already in progress, consider changing " +
                "retry strategy for this type of error. Error: " +
tipe.getErrorMessage());
```

```
       } catch (IdempotentParameterMismatchException ipme) {
           System.out.println("Request rejected because it was retried with a
different payload but with a request token that was already used, " +
               "change request token for this payload to be accepted. Error: " +
ipme.getErrorMessage());
       } catch (Exception e) {
           handleCommonErrors(e);
       }
   }

   private static void handleCommonErrors(Exception exception) {
       try {
           throw exception;
       } catch (InternalServerErrorException isee) {
           System.out.println("Internal Server Error, generally safe to retry with
exponential back-off. Error: " + isee.getErrorMessage());
       } catch (RequestLimitExceededException rlee) {
           System.out.println("Throughput exceeds the current throughput limit for
your account, increase account level throughput before " +
               "retrying. Error: " + rlee.getErrorMessage());
       } catch (ProvisionedThroughputExceededException ptee) {
           System.out.println("Request rate is too high. If you're using a custom
retry strategy make sure to retry with exponential back-off. " +
               "Otherwise consider reducing frequency of requests or increasing
provisioned capacity for your table or secondary index. Error: " +
               ptee.getErrorMessage());
       } catch (ResourceNotFoundException rnfe) {
           System.out.println("One of the tables was not found, verify table exists
before retrying. Error: " + rnfe.getErrorMessage());
       } catch (AmazonServiceException ase) {
           System.out.println("An AmazonServiceException occurred, indicates that
the request was correctly transmitted to the DynamoDB " +
               "service, but for some reason, the service was not able to process
it, and returned an error response instead. Investigate and " +
               "configure retry strategy. Error type: " + ase.getErrorType() + ".
Error message: " + ase.getErrorMessage());
       } catch (AmazonClientException ace) {
           System.out.println("An AmazonClientException occurred, indicates that
the client was unable to get a response from DynamoDB " +
               "service, or the client was unable to parse the response from the
service. Investigate and configure retry strategy. "+
               "Error: " + ace.getMessage());
       } catch (Exception e) {
```

```
            System.out.println("An exception occurred, investigate and configure
 retry strategy. Error: " + e.getMessage());
        }
    }

}
```

The following example shows the different return values when DynamoDB reads items with different conditions specified in the WHERE clause.

AWS CLI

1. Save the following JSON code to a file called partiql.json.

```
[
    // Item exists and projected attribute exists
    {
        "Statement": "SELECT * FROM "Music" WHERE Artist='No One You Know' and
 SongTitle='Call Me Today'"
    },
    // Item exists but projected attributes do not exist
    {
        "Statement": "SELECT non_existent_projected_attribute FROM "Music" WHERE
 Artist='No One You Know' and SongTitle='Call Me Today'"
    },
    // Item does not exist
    {
        "Statement": "SELECT * FROM "Music" WHERE Artist='No One I Know' and
 SongTitle='Call You Today'"
    }
]
```

2. following command in a command prompt.

```
aws dynamodb execute-transaction --transact-statements  file://partiql.json
```

3. The following response is returned:

```
{
    "Responses": [
        // Item exists and projected attribute exists
        {
```

```
            "Item": {
                "Artist":{
                    "S": "No One You Know"
                },
                "SongTitle":{
                    "S": "Call Me Today"
                }
            }
        },
        // Item exists but projected attributes do not exist
        {
            "Item": {}
        },
        // Item does not exist
        {}
    ]
}
```

Running batch operations with PartiQL for DynamoDB

This section describes how to use batch statements with PartiQL for DynamoDB.

> **Note**
>
> - The entire batch must consist of either read statements or write statements; you cannot mix both in one batch.
> - `BatchExecuteStatement` and `BatchWriteItem` can perform no more than 25 statements per batch.
> - `BatchExecuteStatement` makes use of `BatchGetItem` which takes a list of primary keys in separate statements.

Topics

- Syntax
- Parameters
- Examples

Syntax

```
[
  {
    "Statement": "SELECT pk FROM ProblemSet WHERE pk = 'p#9StkWHYTxm7x2AqSXcrfu7' AND
 sk = 'info'"
  },
  {
    "Statement": "SELECT pk FROM ProblemSet WHERE pk = 'p#isC2ChceGbxHgESc4szoTE' AND
 sk = 'info'"
  }
]
```

```
[
   {
      "Statement":" statement ",
      "Parameters":[
         {
            " parametertype " : " parametervalue "
         }, ...]
   } , ...
]
```

Parameters

statement

(Required) A PartiQL for DynamoDB supported statement.

> **Note**
>
> - The entire batch must consist of either read statements or write statements; you cannot mix both in one batch.
> - `BatchExecuteStatement` and `BatchWriteItem` can perform no more than 25 statements per batch.

parametertype

(Optional) A DynamoDB type, if parameters were used when specifying the PartiQL statement.

parametervalue

(Optional) A parameter value if parameters were used when specifying the PartiQL statement.

Examples

AWS CLI

1. Save the following json to a file called partiql.json

```
[
   {
  "Statement": "INSERT INTO Music VALUE {'Artist':?,'SongTitle':?}",
   "Parameters": [{"S": "Acme Band"}, {"S": "Best Song"}]
 },
 {
  "Statement": "UPDATE Music SET AwardsWon=1, AwardDetail={'Grammys':[2020,
 2018]} WHERE Artist='Acme Band' AND SongTitle='PartiQL Rocks'"
    }
]
```

2. Run the following command in a command prompt.

```
aws dynamodb batch-execute-statement  --statements  file://partiql.json
```

Java

```
public class DynamoDBPartiqlBatch {

    public static void main(String[] args) {
        // Create the DynamoDB Client with the region you want
        AmazonDynamoDB dynamoDB = createDynamoDbClient("us-west-2");

        try {
            // Create BatchExecuteStatementRequest
            BatchExecuteStatementRequest batchExecuteStatementRequest =
 createBatchExecuteStatementRequest();
            BatchExecuteStatementResult batchExecuteStatementResult =
 dynamoDB.batchExecuteStatement(batchExecuteStatementRequest);
            System.out.println("BatchExecuteStatement successful.");
            // Handle batchExecuteStatementResult
```

```
        } catch (Exception e) {
            handleBatchExecuteStatementErrors(e);
        }
    }

    private static AmazonDynamoDB createDynamoDbClient(String region) {

        return AmazonDynamoDBClientBuilder.standard().withRegion(region).build();
    }

    private static BatchExecuteStatementRequest createBatchExecuteStatementRequest()
{
        BatchExecuteStatementRequest request = new BatchExecuteStatementRequest();

        // Create statements
        List<BatchStatementRequest> statements = getPartiQLBatchStatements();

        request.setStatements(statements);
        return request;
    }

    private static List<BatchStatementRequest> getPartiQLBatchStatements() {
        List<BatchStatementRequest> statements = new
ArrayList<BatchStatementRequest>();

        statements.add(new BatchStatementRequest()
                               .withStatement("INSERT INTO Music value
{'Artist':'Acme Band','SongTitle':'PartiQL Rocks'}"));

        statements.add(new BatchStatementRequest()
                               .withStatement("UPDATE Music set
AwardDetail.BillBoard=[2020] where Artist='Acme Band' and SongTitle='PartiQL
Rocks'"));

        return statements;
    }

    // Handles errors during BatchExecuteStatement execution. Use recommendations in
error messages below to add error handling specific to
    // your application use-case.
    private static void handleBatchExecuteStatementErrors(Exception exception) {
        try {
            throw exception;
```

```
        } catch (Exception e) {
            // There are no API specific errors to handle for BatchExecuteStatement,
common DynamoDB API errors are handled below
            handleCommonErrors(e);
        }
    }

    private static void handleCommonErrors(Exception exception) {
        try {
            throw exception;
        } catch (InternalServerErrorException isee) {
            System.out.println("Internal Server Error, generally safe to retry with
exponential back-off. Error: " + isee.getErrorMessage());
        } catch (RequestLimitExceededException rlee) {
            System.out.println("Throughput exceeds the current throughput limit for
your account, increase account level throughput before " +
                "retrying. Error: " + rlee.getErrorMessage());
        } catch (ProvisionedThroughputExceededException ptee) {
            System.out.println("Request rate is too high. If you're using a custom
retry strategy make sure to retry with exponential back-off. " +
                "Otherwise consider reducing frequency of requests or increasing
provisioned capacity for your table or secondary index. Error: " +
                ptee.getErrorMessage());
        } catch (ResourceNotFoundException rnfe) {
            System.out.println("One of the tables was not found, verify table exists
before retrying. Error: " + rnfe.getErrorMessage());
        } catch (AmazonServiceException ase) {
            System.out.println("An AmazonServiceException occurred, indicates that
the request was correctly transmitted to the DynamoDB " +
                "service, but for some reason, the service was not able to process
it, and returned an error response instead. Investigate and " +
                "configure retry strategy. Error type: " + ase.getErrorType() + ".
Error message: " + ase.getErrorMessage());
        } catch (AmazonClientException ace) {
            System.out.println("An AmazonClientException occurred, indicates that
the client was unable to get a response from DynamoDB " +
                "service, or the client was unable to parse the response from the
service. Investigate and configure retry strategy. "+
                "Error: " + ace.getMessage());
        } catch (Exception e) {
            System.out.println("An exception occurred, investigate and configure
retry strategy. Error: " + e.getMessage());
        }
    }
```

```
}
```

IAM security policies with PartiQL for DynamoDB

The following permissions are required:

- To read items using PartiQL for DynamoDB, you must have `dynamodb:PartiQLSelect` permission on the table or index.
- To insert items using PartiQL for DynamoDB, you must have `dynamodb:PartiQLInsert` permission on the table or index.
- To update items using PartiQL for DynamoDB, you must have `dynamodb:PartiQLUpdate` permission on the table or index.
- To delete items using PartiQL for DynamoDB, you must have `dynamodb:PartiQLDelete` permission on the table or index.

Example: Allow all PartiQL for DynamoDB statements (Select/Insert/Update/Delete) on a table

The following IAM policy grants permissions to run all PartiQL for DynamoDB statements on a table.

JSON

```
{
   "Version":"2012-10-17",
   "Statement":[
      {
         "Effect":"Allow",
         "Action":[
            "dynamodb:PartiQLInsert",
            "dynamodb:PartiQLUpdate",
            "dynamodb:PartiQLDelete",
            "dynamodb:PartiQLSelect"
         ],
         "Resource":[
            "arn:aws:dynamodb:us-west-2:123456789012:table/Music"
         ]
      }
```

```
   ]
}
```

Example: Allow PartiQL for DynamoDB select statements on a table

The following IAM policy grants permissions to run the `select` statement on a specific table.

JSON

```
{
   "Version":"2012-10-17",
   "Statement":[
      {
         "Effect":"Allow",
         "Action":[
            "dynamodb:PartiQLSelect"
         ],
         "Resource":[
            "arn:aws:dynamodb:us-west-2:123456789012:table/Music"
         ]
      }
   ]
}
```

Example: Allow PartiQL for DynamoDB insert statements on an index

The following IAM policy grants permissions to run the `insert` statement on a specific index.

JSON

```
{
   "Version":"2012-10-17",
   "Statement":[
      {
         "Effect":"Allow",
         "Action":[
            "dynamodb:PartiQLInsert"
         ],
         "Resource":[
            "arn:aws:dynamodb:us-west-2:123456789012:table/Music/index/index1"
```

```
         ]
      }
   ]
}
```

Example: Allow PartiQL for DynamoDB transactional statements only on a table

The following IAM policy grants permissions to run only transactional statements on a specific table.

JSON

```
{
   "Version":"2012-10-17",
   "Statement":[
      {
         "Effect":"Allow",
         "Action":[
            "dynamodb:PartiQLInsert",
            "dynamodb:PartiQLUpdate",
            "dynamodb:PartiQLDelete",
            "dynamodb:PartiQLSelect"
         ],
         "Resource":[
            "arn:aws:dynamodb:us-west-2:123456789012:table/Music"
         ],
         "Condition":{
            "StringEquals":{
               "dynamodb:EnclosingOperation":[
                  "ExecuteTransaction"
               ]
            }
         }
      }
   ]
}
```

Example: Allow PartiQL for DynamoDB non-transactional reads and writes and block PartiQL transactional reads and writes transactional statements on a table.

The following IAM policy grants permissions to run PartiQL for DynamoDB non-transactional reads and writes while blocking PartiQL for DynamoDB transactional reads and writes.

JSON

```
{
   "Version":"2012-10-17",
   "Statement":[
      {
         "Effect":"Deny",
         "Action":[
            "dynamodb:PartiQLInsert",
            "dynamodb:PartiQLUpdate",
            "dynamodb:PartiQLDelete",
            "dynamodb:PartiQLSelect"
         ],
         "Resource":[
            "arn:aws:dynamodb:us-west-2:123456789012:table/Music"
         ],
         "Condition":{
            "StringEquals":{
               "dynamodb:EnclosingOperation":[
                  "ExecuteTransaction"
               ]
            }
         }
      },
      {
         "Effect":"Allow",
         "Action":[
            "dynamodb:PartiQLInsert",
            "dynamodb:PartiQLUpdate",
            "dynamodb:PartiQLDelete",
            "dynamodb:PartiQLSelect"
         ],
         "Resource":[
            "arn:aws:dynamodb:us-west-2:123456789012:table/Music"
         ]
      }
   ]
```

```
}
```

Example: Allow select statements and deny full table scan statements in PartiQL for DynamoDB

The following IAM policy grants permissions to run the `select` statement on a specific table while blocking `select` statements that result in a full table scan.

JSON

```
{
   "Version":"2012-10-17",
   "Statement":[
      {
         "Effect":"Deny",
         "Action":[
            "dynamodb:PartiQLSelect"
         ],
         "Resource":[
            "arn:aws:dynamodb:us-west-2:123456789012:table/WatchList"
         ],
         "Condition":{
            "Bool":{
               "dynamodb:FullTableScan":[
                  "true"
               ]
            }
         }
      },
      {
         "Effect":"Allow",
         "Action":[
            "dynamodb:PartiQLSelect"
         ],
         "Resource":[
            "arn:aws:dynamodb:us-west-2:123456789012:table/WatchList"
         ]
      }
   ]
}
```

Working with items: Java

You can use the AWS SDK for Java Document API to perform typical create, read, update, and delete (CRUD) operations on Amazon DynamoDB items in a table.

> **Note**
>
> The SDK for Java also provides an object persistence model, allowing you to map your client-side classes to DynamoDB tables. This approach can reduce the amount of code that you have to write. For more information, see Java 1.x: DynamoDBMapper.

This section contains Java examples to perform several Java Document API item actions and several complete working examples.

Topics

- Putting an item
- Getting an item
- Batch write: Putting and deleting multiple items
- Batch get: Getting multiple items
- Updating an item
- Deleting an item
- Example: CRUD operations using the AWS SDK for Java document API
- Example: Batch operations using AWS SDK for Java document API
- Example: Handling binary type attributes using the AWS SDK for Java document API

Putting an item

The `putItem` method stores an item in a table. If the item exists, it replaces the entire item. Instead of replacing the entire item, if you want to update only specific attributes, you can use the `updateItem` method. For more information, see Updating an item.

Java v2

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
```

```
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.PutItemRequest;
import software.amazon.awssdk.services.dynamodb.model.PutItemResponse;
import software.amazon.awssdk.services.dynamodb.model.ResourceNotFoundException;
import java.util.HashMap;

/**
 * Before running this Java V2 code example, set up your development
 * environment, including your credentials.
 *
 * For more information, see the following documentation topic:
 *
 * https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-started.html
 *
 * To place items into an Amazon DynamoDB table using the AWS SDK for Java V2,
 * its better practice to use the
 * Enhanced Client. See the EnhancedPutItem example.
 */
public class PutItem {
    public static void main(String[] args) {
        final String usage = """

                Usage:
                    <tableName> <key> <keyVal> <albumtitle> <albumtitleval> <awards>
 <awardsval> <Songtitle> <songtitleval>

                Where:
                    tableName - The Amazon DynamoDB table in which an item is placed
 (for example, Music3).
                    key - The key used in the Amazon DynamoDB table (for example,
 Artist).
                    keyval - The key value that represents the item to get (for
 example, Famous Band).
                    albumTitle - The Album title (for example, AlbumTitle).
                    AlbumTitleValue - The name of the album (for example, Songs
 About Life ).
                    Awards - The awards column (for example, Awards).
                    AwardVal - The value of the awards (for example, 10).
                    SongTitle - The song title (for example, SongTitle).
                    SongTitleVal - The value of the song title (for example, Happy
 Day).
                **Warning** This program will  place an item that you specify into a
 table!
                """;
```

```
        if (args.length != 9) {
            System.out.println(usage);
            System.exit(1);
        }

        String tableName = args[0];
        String key = args[1];
        String keyVal = args[2];
        String albumTitle = args[3];
        String albumTitleValue = args[4];
        String awards = args[5];
        String awardVal = args[6];
        String songTitle = args[7];
        String songTitleVal = args[8];

        Region region = Region.US_EAST_1;
        DynamoDbClient ddb = DynamoDbClient.builder()
                .region(region)
                .build();

        putItemInTable(ddb, tableName, key, keyVal, albumTitle, albumTitleValue,
awards, awardVal, songTitle,
                songTitleVal);
        System.out.println("Done!");
        ddb.close();
    }

    public static void putItemInTable(DynamoDbClient ddb,
            String tableName,
            String key,
            String keyVal,
            String albumTitle,
            String albumTitleValue,
            String awards,
            String awardVal,
            String songTitle,
            String songTitleVal) {

        HashMap<String, AttributeValue> itemValues = new HashMap<>();
        itemValues.put(key, AttributeValue.builder().s(keyVal).build());
        itemValues.put(songTitle, AttributeValue.builder().s(songTitleVal).build());
        itemValues.put(albumTitle,
AttributeValue.builder().s(albumTitleValue).build());
```

```
        itemValues.put(awards, AttributeValue.builder().s(awardVal).build());

        PutItemRequest request = PutItemRequest.builder()
                .tableName(tableName)
                .item(itemValues)
                .build();

        try {
            PutItemResponse response = ddb.putItem(request);
            System.out.println(tableName + " was successfully updated. The request
 id is "
                    + response.responseMetadata().requestId());

        } catch (ResourceNotFoundException e) {
            System.err.format("Error: The Amazon DynamoDB table \"%s\" can't be
 found.\n", tableName);
            System.err.println("Be sure that it exists and that you've typed its
 name correctly!");
            System.exit(1);
        } catch (DynamoDbException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }
}
```

Java v1

Follow these steps:

1. Create an instance of the DynamoDB class.
2. Create an instance of the `Table` class to represent the table you want to work with.
3. Create an instance of the `Item` class to represent the new item. You must specify the new item's primary key and its attributes.
4. Call the `putItem` method of the `Table` object, using the `Item` that you created in the preceding step.

The following Java code example demonstrates the preceding tasks. The code writes a new item to the `ProductCatalog` table.

Example

```
AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();
DynamoDB dynamoDB = new DynamoDB(client);

Table table = dynamoDB.getTable("ProductCatalog");

// Build a list of related items
List<Number> relatedItems = new ArrayList<Number>();
relatedItems.add(341);
relatedItems.add(472);
relatedItems.add(649);

//Build a map of product pictures
Map<String, String> pictures = new HashMap<String, String>();
pictures.put("FrontView", "http://example.com/products/123_front.jpg");
pictures.put("RearView", "http://example.com/products/123_rear.jpg");
pictures.put("SideView", "http://example.com/products/123_left_side.jpg");

//Build a map of product reviews
Map<String, List<String>> reviews = new HashMap<String, List<String>>();

List<String> fiveStarReviews = new ArrayList<String>();
fiveStarReviews.add("Excellent! Can't recommend it highly enough!  Buy it!");
fiveStarReviews.add("Do yourself a favor and buy this");
reviews.put("FiveStar", fiveStarReviews);

List<String> oneStarReviews = new ArrayList<String>();
oneStarReviews.add("Terrible product!  Do not buy this.");
reviews.put("OneStar", oneStarReviews);

// Build the item
Item item = new Item()
    .withPrimaryKey("Id", 123)
    .withString("Title", "Bicycle 123")
    .withString("Description", "123 description")
    .withString("BicycleType", "Hybrid")
    .withString("Brand", "Brand-Company C")
    .withNumber("Price", 500)
    .withStringSet("Color",  new HashSet<String>(Arrays.asList("Red", "Black")))
    .withString("ProductCategory", "Bicycle")
    .withBoolean("InStock", true)
    .withNull("QuantityOnHand")
    .withList("RelatedItems", relatedItems)
```

```
    .withMap("Pictures", pictures)
    .withMap("Reviews", reviews);

// Write the item to the table
PutItemOutcome outcome = table.putItem(item);
```

In the preceding example, the item has attributes that are scalars (`String`, `Number`, `Boolean`, `Null`), sets (`String Set`), and document types (`List`, `Map`).

Specifying optional parameters

Along with the required parameters, you can also specify optional parameters to the `putItem` method. For example, the following Java code example uses an optional parameter to specify a condition for uploading the item. If the condition you specify is not met, the AWS SDK for Java throws a `ConditionalCheckFailedException`. The code example specifies the following optional parameters in the `putItem` method:

- A `ConditionExpression` that defines the conditions for the request. The code defines the condition that the existing item with the same primary key is replaced only if it has an ISBN attribute that equals a specific value.
- A map for `ExpressionAttributeValues` that is used in the condition. In this case, there is only one substitution required: The placeholder `:val` in the condition expression is replaced at runtime with the actual ISBN value to be checked.

The following example adds a new book item using these optional parameters.

Example

```
Item item = new Item()
    .withPrimaryKey("Id", 104)
    .withString("Title", "Book 104 Title")
    .withString("ISBN", "444-4444444444")
    .withNumber("Price", 20)
    .withStringSet("Authors",
        new HashSet<String>(Arrays.asList("Author1", "Author2")));

Map<String, Object> expressionAttributeValues = new HashMap<String, Object>();
expressionAttributeValues.put(":val", "444-4444444444");
```

```
PutItemOutcome outcome = table.putItem(
    item,
    "ISBN = :val", // ConditionExpression parameter
    null,          // ExpressionAttributeNames parameter - we're not using it for this
 example
    expressionAttributeValues);
```

PutItem and JSON documents

You can store a JSON document as an attribute in a DynamoDB table. To do this, use the `withJSON` method of `Item`. This method parses the JSON document and maps each element to a native DynamoDB data type.

Suppose that you wanted to store the following JSON document, containing vendors that can fulfill orders for a particular product.

Example

```
{
    "V01": {
        "Name": "Acme Books",
        "Offices": [ "Seattle" ]
    },
    "V02": {
        "Name": "New Publishers, Inc.",
        "Offices": ["London", "New York"
        ]
    },
    "V03": {
        "Name": "Better Buy Books",
        "Offices": [ "Tokyo", "Los Angeles", "Sydney"
        ]
    }
}
```

You can use the `withJSON` method to store this in the `ProductCatalog` table, in a `Map` attribute named `VendorInfo`. The following Java code example demonstrates how to do this.

```
// Convert the document into a String.  Must escape all double-quotes.
String vendorDocument = "{"
    + "    \"V01\": {"
```

```
    + "        \"Name\": \"Acme Books\","
    + "        \"Offices\": [ \"Seattle\" ]"
    + "    },"
    + "    \"V02\": {"
    + "        \"Name\": \"New Publishers, Inc.\","
    + "        \"Offices\": [ \"London\", \"New York\"" + "]" + "},"
    + "    \"V03\": {"
    + "        \"Name\": \"Better Buy Books\","
    +          "\"Offices\": [ \"Tokyo\", \"Los Angeles\", \"Sydney\""
    + "            ]"
    + "        }"
    + "    }";

Item item = new Item()
    .withPrimaryKey("Id", 210)
    .withString("Title", "Book 210 Title")
    .withString("ISBN", "210-2102102102")
    .withNumber("Price", 30)
    .withJSON("VendorInfo", vendorDocument);

PutItemOutcome outcome = table.putItem(item);
```

Getting an item

To retrieve a single item, use the `getItem` method of a `Table` object. Follow these steps:

1. Create an instance of the DynamoDB class.
2. Create an instance of the `Table` class to represent the table you want to work with.
3. Call the `getItem` method of the `Table` instance. You must specify the primary key of the item that you want to retrieve.

The following Java code example demonstrates the preceding steps. The code gets the item that has the specified partition key.

```
AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();
DynamoDB dynamoDB = new DynamoDB(client);

Table table = dynamoDB.getTable("ProductCatalog");

Item item = table.getItem("Id", 210);
```

Specifying optional parameters

Along with the required parameters, you can also specify optional parameters for the `getItem` method. For example, the following Java code example uses an optional method to retrieve only a specific list of attributes and to specify strongly consistent reads. (To learn more about read consistency, see DynamoDB read consistency.)

You can use a `ProjectionExpression` to retrieve only specific attributes or elements, rather than an entire item. A `ProjectionExpression` can specify top-level or nested attributes using document paths. For more information, see Using projection expressions in DynamoDB.

The parameters of the `getItem` method don't let you specify read consistency. However, you can create a `GetItemSpec`, which provides full access to all of the inputs to the low-level `GetItem` operation. The following code example creates a `GetItemSpec` and uses that spec as input to the `getItem` method.

Example

```
GetItemSpec spec = new GetItemSpec()
    .withPrimaryKey("Id", 206)
    .withProjectionExpression("Id, Title, RelatedItems[0], Reviews.FiveStar")
    .withConsistentRead(true);

Item item = table.getItem(spec);

System.out.println(item.toJSONPretty());
```

To print an `Item` in a human-readable format, use the `toJSONPretty` method. The output from the previous example looks like the following.

```
{
  "RelatedItems" : [ 341 ],
  "Reviews" : {
    "FiveStar" : [ "Excellent! Can't recommend it highly enough! Buy it!", "Do yourself
 a favor and buy this" ]
  },
  "Id" : 123,
  "Title" : "20-Bicycle 123"
}
```

GetItem and JSON documents

In the PutItem and JSON documents section, you store a JSON document in a `Map` attribute named `VendorInfo`. You can use the `getItem` method to retrieve the entire document in JSON format. Or you can use document path notation to retrieve only some of the elements in the document. The following Java code example demonstrates these techniques.

```
GetItemSpec spec = new GetItemSpec()
    .withPrimaryKey("Id", 210);

System.out.println("All vendor info:");
spec.withProjectionExpression("VendorInfo");
System.out.println(table.getItem(spec).toJSON());

System.out.println("A single vendor:");
spec.withProjectionExpression("VendorInfo.V03");
System.out.println(table.getItem(spec).toJSON());

System.out.println("First office location for this vendor:");
spec.withProjectionExpression("VendorInfo.V03.Offices[0]");
System.out.println(table.getItem(spec).toJSON());
```

The output from the previous example looks like the following.

```
All vendor info:
{"VendorInfo":{"V03":{"Name":"Better Buy Books","Offices":["Tokyo","Los
 Angeles","Sydney"]},"V02":{"Name":"New Publishers, Inc.","Offices":["London","New
 York"]},"V01":{"Name":"Acme Books","Offices":["Seattle"]}}}
A single vendor:
{"VendorInfo":{"V03":{"Name":"Better Buy Books","Offices":["Tokyo","Los
 Angeles","Sydney"]}}}
First office location for a single vendor:
{"VendorInfo":{"V03":{"Offices":["Tokyo"]}}}
```

> **Note**
>
> You can use the `toJSON` method to convert any item (or its attributes) to a JSON-formatted string. The following code retrieves several top-level and nested attributes and prints the results as JSON.

```
GetItemSpec spec = new GetItemSpec()
    .withPrimaryKey("Id", 210)
    .withProjectionExpression("VendorInfo.V01, Title, Price");

Item item = table.getItem(spec);
System.out.println(item.toJSON());
```

The output looks like the following.

```
{"VendorInfo":{"V01":{"Name":"Acme Books","Offices":
["Seattle"]}},"Price":30,"Title":"Book 210 Title"}
```

Batch write: Putting and deleting multiple items

Batch write refers to putting and deleting multiple items in a batch. The `batchWriteItem` method enables you to put and delete multiple items from one or more tables in a single call. The following are the steps to put or delete multiple items using the AWS SDK for Java Document API.

1. Create an instance of the `DynamoDB` class.
2. Create an instance of the `TableWriteItems` class that describes all the put and delete operations for a table. If you want to write to multiple tables in a single batch write operation, you must create one `TableWriteItems` instance per table.
3. Call the `batchWriteItem` method by providing the `TableWriteItems` objects that you created in the preceding step.
4. Process the response. You should check if there were any unprocessed request items returned in the response. This could happen if you reach the provisioned throughput quota or some other transient error. Also, DynamoDB limits the request size and the number of operations you can specify in a request. If you exceed these limits, DynamoDB rejects the request. For more information, see Quotas in Amazon DynamoDB.

The following Java code example demonstrates the preceding steps. The example performs a `batchWriteItem` operation on two tables: `Forum` and `Thread`. The corresponding `TableWriteItems` objects define the following actions:

- Put an item in the `Forum` table.

- Put and delete an item in the `Thread` table.

The code then calls `batchWriteItem` to perform the operation.

```
AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();
DynamoDB dynamoDB = new DynamoDB(client);

TableWriteItems forumTableWriteItems = new TableWriteItems("Forum")
    .withItemsToPut(
        new Item()
            .withPrimaryKey("Name", "Amazon RDS")
            .withNumber("Threads", 0));

TableWriteItems threadTableWriteItems = new TableWriteItems("Thread")
    .withItemsToPut(
        new Item()
            .withPrimaryKey("ForumName","Amazon RDS","Subject","Amazon RDS Thread 1")
    .withHashAndRangeKeysToDelete("ForumName","Some partition key value", "Amazon S3",
 "Some sort key value");

BatchWriteItemOutcome outcome = dynamoDB.batchWriteItem(forumTableWriteItems,
 threadTableWriteItems);

// Code for checking unprocessed items is omitted in this example
```

For a working example, see Example: Batch write operation using the AWS SDK for Java document API.

Batch get: Getting multiple items

The `batchGetItem` method enables you to retrieve multiple items from one or more tables. To retrieve a single item, you can use the `getItem` method.

Follow these steps:

1. Create an instance of the `DynamoDB` class.
2. Create an instance of the `TableKeysAndAttributes` class that describes a list of primary key values to retrieve from a table. If you want to read from multiple tables in a single batch get operation, you must create one `TableKeysAndAttributes` instance per table.
3. Call the `batchGetItem` method by providing the `TableKeysAndAttributes` objects that you created in the preceding step.

The following Java code example demonstrates the preceding steps. The example retrieves two items from the `Forum` table and three items from the `Thread` table.

```
AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();
DynamoDB dynamoDB = new DynamoDB(client);

    TableKeysAndAttributes forumTableKeysAndAttributes = new
 TableKeysAndAttributes(forumTableName);
    forumTableKeysAndAttributes.addHashOnlyPrimaryKeys("Name",
    "Amazon S3",
    "Amazon DynamoDB");

TableKeysAndAttributes threadTableKeysAndAttributes = new
 TableKeysAndAttributes(threadTableName);
threadTableKeysAndAttributes.addHashAndRangePrimaryKeys("ForumName", "Subject",
    "Amazon DynamoDB","DynamoDB Thread 1",
    "Amazon DynamoDB","DynamoDB Thread 2",
    "Amazon S3","S3 Thread 1");

BatchGetItemOutcome outcome = dynamoDB.batchGetItem(
    forumTableKeysAndAttributes, threadTableKeysAndAttributes);

for (String tableName : outcome.getTableItems().keySet()) {
    System.out.println("Items in table " + tableName);
    List<Item> items = outcome.getTableItems().get(tableName);
    for (Item item : items) {
        System.out.println(item);
    }
}
```

Specifying optional parameters

Along with the required parameters, you can also specify optional parameters when using `batchGetItem`. For example, you can provide a `ProjectionExpression` with each `TableKeysAndAttributes` you define. This allows you to specify the attributes that you want to retrieve from the table.

The following code example retrieves two items from the `Forum` table. The `withProjectionExpression` parameter specifies that only the `Threads` attribute is to be retrieved.

Example

```
TableKeysAndAttributes forumTableKeysAndAttributes = new
 TableKeysAndAttributes("Forum")
    .withProjectionExpression("Threads");

forumTableKeysAndAttributes.addHashOnlyPrimaryKeys("Name",
    "Amazon S3",
    "Amazon DynamoDB");

BatchGetItemOutcome outcome = dynamoDB.batchGetItem(forumTableKeysAndAttributes);
```

Updating an item

The `updateItem` method of a `Table` object can update existing attribute values, add new attributes, or delete attributes from an existing item.

The `updateItem` method behaves as follows:

- If an item does not exist (no item in the table with the specified primary key), `updateItem` adds a new item to the table.
- If an item exists, `updateItem` performs the update as specified by the `UpdateExpression` parameter.

> **Note**
>
> It is also possible to "update" an item using `putItem`. For example, if you call `putItem` to add an item to the table, but there is already an item with the specified primary key, `putItem` replaces the entire item. If there are attributes in the existing item that are not specified in the input, `putItem` removes those attributes from the item.
> In general, we recommend that you use `updateItem` whenever you want to modify any item attributes. The `updateItem` method only modifies the item attributes that you specify in the input, and the other attributes in the item remain unchanged.

Follow these steps:

1. Create an instance of the `Table` class to represent the table that you want to work with.

2. Call the `updateTable` method of the `Table` instance. You must specify the primary key of the item that you want to retrieve, along with an `UpdateExpression` that describes the attributes to modify and how to modify them.

The following Java code example demonstrates the preceding tasks. The code updates a book item in the `ProductCatalog` table. It adds a new author to the set of `Authors` and deletes the existing `ISBN` attribute. It also reduces the price by one.

An `ExpressionAttributeValues` map is used in the `UpdateExpression`. The placeholders `:val1` and `:val2` are replaced at runtime with the actual values for `Authors` and `Price`.

```
AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();
DynamoDB dynamoDB = new DynamoDB(client);

Table table = dynamoDB.getTable("ProductCatalog");

Map<String, String> expressionAttributeNames = new HashMap<String, String>();
expressionAttributeNames.put("#A", "Authors");
expressionAttributeNames.put("#P", "Price");
expressionAttributeNames.put("#I", "ISBN");

Map<String, Object> expressionAttributeValues = new HashMap<String, Object>();
expressionAttributeValues.put(":val1",
    new HashSet<String>(Arrays.asList("Author YY","Author ZZ")));
expressionAttributeValues.put(":val2", 1);   //Price

UpdateItemOutcome outcome =  table.updateItem(
    "Id",          // key attribute name
    101,           // key attribute value
    "add #A :val1 set #P = #P - :val2 remove #I", // UpdateExpression
    expressionAttributeNames,
    expressionAttributeValues);
```

Specifying optional parameters

Along with the required parameters, you can also specify optional parameters for the `updateItem` method, including a condition that must be met in order for the update is to occur. If the condition you specify is not met, the AWS SDK for Java throws a `ConditionalCheckFailedException`. For example, the following Java code example conditionally updates a book item price to 25. It specifies a `ConditionExpression` stating that the price should be updated only if the existing price is 20.

Example

```
Table table = dynamoDB.getTable("ProductCatalog");

Map<String, String> expressionAttributeNames = new HashMap<String, String>();
expressionAttributeNames.put("#P", "Price");

Map<String, Object> expressionAttributeValues = new HashMap<String, Object>();
expressionAttributeValues.put(":val1", 25);  // update Price to 25...
expressionAttributeValues.put(":val2", 20);  //...but only if existing Price is 20

UpdateItemOutcome outcome = table.updateItem(
    new PrimaryKey("Id",101),
    "set #P = :val1", // UpdateExpression
    "#P = :val2",     // ConditionExpression
    expressionAttributeNames,
    expressionAttributeValues);
```

Atomic counter

You can use `updateItem` to implement an atomic counter, where you increment or decrement the value of an existing attribute without interfering with other write requests. To increment an atomic counter, use an `UpdateExpression` with a `set` action to add a numeric value to an existing attribute of type `Number`.

The following example demonstrates this, incrementing the `Quantity` attribute by one. It also demonstrates the use of the `ExpressionAttributeNames` parameter in an `UpdateExpression`.

```
Table table = dynamoDB.getTable("ProductCatalog");

Map<String,String> expressionAttributeNames = new HashMap<String,String>();
expressionAttributeNames.put("#p", "PageCount");

Map<String,Object> expressionAttributeValues = new HashMap<String,Object>();
expressionAttributeValues.put(":val", 1);

UpdateItemOutcome outcome = table.updateItem(
    "Id", 121,
    "set #p = #p + :val",
    expressionAttributeNames,
    expressionAttributeValues);
```

Deleting an item

The `deleteItem` method deletes an item from a table. You must provide the primary key of the item that you want to delete.

Follow these steps:

1. Create an instance of the `DynamoDB` client.
2. Call the `deleteItem` method by providing the key of the item you want to delete.

The following Java example demonstrates these tasks.

Example

```
AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();
DynamoDB dynamoDB = new DynamoDB(client);

Table table = dynamoDB.getTable("ProductCatalog");

DeleteItemOutcome outcome = table.deleteItem("Id", 101);
```

Specifying optional parameters

You can specify optional parameters for `deleteItem`. For example, the following Java code example specifies a `ConditionExpression`, stating that a book item in `ProductCatalog` can only be deleted if the book is no longer in publication (the `InPublication` attribute is false).

Example

```
Map<String,Object> expressionAttributeValues = new HashMap<String,Object>();
expressionAttributeValues.put(":val", false);

DeleteItemOutcome outcome = table.deleteItem("Id",103,
    "InPublication = :val",
    null, // ExpressionAttributeNames - not used in this example
    expressionAttributeValues);
```

Example: CRUD operations using the AWS SDK for Java document API

The following code example illustrates CRUD operations on an Amazon DynamoDB item. The example creates an item, retrieves it, performs various updates, and finally deletes the item.

Note

The SDK for Java also provides an object persistence model, enabling you to map your client-side classes to DynamoDB tables. This approach can reduce the amount of code that you have to write. For more information, see Java 1.x: DynamoDBMapper.

Note

This code example assumes that you have already loaded data into DynamoDB for your account by following the instructions in the Creating tables and loading data for code examples in DynamoDB section.
For step-by-step instructions to run the following example, see Java code examples.

```
package com.amazonaws.codesamples.document;

import java.io.IOException;
import java.util.Arrays;
import java.util.HashMap;
import java.util.HashSet;
import java.util.Map;

import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;
import com.amazonaws.services.dynamodbv2.document.DeleteItemOutcome;
import com.amazonaws.services.dynamodbv2.document.DynamoDB;
import com.amazonaws.services.dynamodbv2.document.Item;
import com.amazonaws.services.dynamodbv2.document.Table;
import com.amazonaws.services.dynamodbv2.document.UpdateItemOutcome;
import com.amazonaws.services.dynamodbv2.document.spec.DeleteItemSpec;
import com.amazonaws.services.dynamodbv2.document.spec.UpdateItemSpec;
import com.amazonaws.services.dynamodbv2.document.utils.NameMap;
import com.amazonaws.services.dynamodbv2.document.utils.ValueMap;
import com.amazonaws.services.dynamodbv2.model.ReturnValue;

public class DocumentAPIItemCRUDExample {

    static AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();
    static DynamoDB dynamoDB = new DynamoDB(client);
```

```
    static String tableName = "ProductCatalog";

    public static void main(String[] args) throws IOException {

        createItems();

        retrieveItem();

        // Perform various updates.
        updateMultipleAttributes();
        updateAddNewAttribute();
        updateExistingAttributeConditionally();

        // Delete the item.
        deleteItem();

    }

    private static void createItems() {

        Table table = dynamoDB.getTable(tableName);
        try {

            Item item = new Item().withPrimaryKey("Id", 120).withString("Title", "Book
120 Title")
                    .withString("ISBN", "120-1111111111")
                    .withStringSet("Authors", new
HashSet<String>(Arrays.asList("Author12", "Author22")))
                    .withNumber("Price", 20).withString("Dimensions",
"8.5x11.0x.75").withNumber("PageCount", 500)
                    .withBoolean("InPublication", false).withString("ProductCategory",
"Book");
            table.putItem(item);

            item = new Item().withPrimaryKey("Id", 121).withString("Title", "Book 121
Title")
                    .withString("ISBN", "121-1111111111")
                    .withStringSet("Authors", new
HashSet<String>(Arrays.asList("Author21", "Author 22")))
                    .withNumber("Price", 20).withString("Dimensions",
"8.5x11.0x.75").withNumber("PageCount", 500)
                    .withBoolean("InPublication", true).withString("ProductCategory",
"Book");
```

```
            table.putItem(item);

        } catch (Exception e) {
            System.err.println("Create items failed.");
            System.err.println(e.getMessage());

        }
    }

    private static void retrieveItem() {
        Table table = dynamoDB.getTable(tableName);

        try {

            Item item = table.getItem("Id", 120, "Id, ISBN, Title, Authors", null);

            System.out.println("Printing item after retrieving it....");
            System.out.println(item.toJSONPretty());

        } catch (Exception e) {
            System.err.println("GetItem failed.");
            System.err.println(e.getMessage());
        }

    }

    private static void updateAddNewAttribute() {
        Table table = dynamoDB.getTable(tableName);

        try {

            UpdateItemSpec updateItemSpec = new UpdateItemSpec().withPrimaryKey("Id",
121)
                    .withUpdateExpression("set #na = :val1").withNameMap(new
NameMap().with("#na", "NewAttribute"))
                    .withValueMap(new ValueMap().withString(":val1", "Some value"))
                    .withReturnValues(ReturnValue.ALL_NEW);

            UpdateItemOutcome outcome = table.updateItem(updateItemSpec);

            // Check the response.
            System.out.println("Printing item after adding new attribute...");
            System.out.println(outcome.getItem().toJSONPretty());
```

```
        } catch (Exception e) {
            System.err.println("Failed to add new attribute in " + tableName);
            System.err.println(e.getMessage());
        }
    }

    private static void updateMultipleAttributes() {

        Table table = dynamoDB.getTable(tableName);

        try {

            UpdateItemSpec updateItemSpec = new UpdateItemSpec().withPrimaryKey("Id",
120)
                    .withUpdateExpression("add #a :val1 set #na=:val2")
                    .withNameMap(new NameMap().with("#a", "Authors").with("#na",
"NewAttribute"))
                    .withValueMap(
                            new ValueMap().withStringSet(":val1", "Author YY", "Author
ZZ").withString(":val2",
                                    "someValue"))
                    .withReturnValues(ReturnValue.ALL_NEW);

            UpdateItemOutcome outcome = table.updateItem(updateItemSpec);

            // Check the response.
            System.out.println("Printing item after multiple attribute update...");
            System.out.println(outcome.getItem().toJSONPretty());

        } catch (Exception e) {
            System.err.println("Failed to update multiple attributes in " + tableName);
            System.err.println(e.getMessage());

        }
    }

    private static void updateExistingAttributeConditionally() {

        Table table = dynamoDB.getTable(tableName);

        try {

            // Specify the desired price (25.00) and also the condition (price =
            // 20.00)
```

```
            UpdateItemSpec updateItemSpec = new UpdateItemSpec().withPrimaryKey("Id",
120)
                    .withReturnValues(ReturnValue.ALL_NEW).withUpdateExpression("set #p
= :val1")
                    .withConditionExpression("#p = :val2").withNameMap(new
NameMap().with("#p", "Price"))
                    .withValueMap(new ValueMap().withNumber(":val1",
25).withNumber(":val2", 20));

            UpdateItemOutcome outcome = table.updateItem(updateItemSpec);

            // Check the response.
            System.out.println("Printing item after conditional update to new
attribute...");
            System.out.println(outcome.getItem().toJSONPretty());

        } catch (Exception e) {
            System.err.println("Error updating item in " + tableName);
            System.err.println(e.getMessage());
        }
    }

    private static void deleteItem() {

        Table table = dynamoDB.getTable(tableName);

        try {

            DeleteItemSpec deleteItemSpec = new DeleteItemSpec().withPrimaryKey("Id",
120)
                    .withConditionExpression("#ip = :val").withNameMap(new
NameMap().with("#ip", "InPublication"))
                    .withValueMap(new ValueMap().withBoolean(":val",
false)).withReturnValues(ReturnValue.ALL_OLD);

            DeleteItemOutcome outcome = table.deleteItem(deleteItemSpec);

            // Check the response.
            System.out.println("Printing item that was deleted...");
            System.out.println(outcome.getItem().toJSONPretty());

        } catch (Exception e) {
            System.err.println("Error deleting item in " + tableName);
```

```
            System.err.println(e.getMessage());
        }
    }
}
```

Example: Batch operations using AWS SDK for Java document API

This section provides examples of batch write and batch get operations in Amazon DynamoDB using the AWS SDK for Java Document API.

> **Note**
>
> The SDK for Java also provides an object persistence model, enabling you to map your client-side classes to DynamoDB tables. This approach can reduce the amount of code that you have to write. For more information, see Java 1.x: DynamoDBMapper.

Topics

- Example: Batch write operation using the AWS SDK for Java document API
- Example: Batch get operation using the AWS SDK for Java document API

Example: Batch write operation using the AWS SDK for Java document API

The following Java code example uses the `batchWriteItem` method to perform the following put and delete operations:

- Put one item in the `Forum` table.
- Put one item and delete one item from the `Thread` table.

You can specify any number of put and delete requests against one or more tables when creating your batch write request. However, `batchWriteItem` limits the size of a batch write request and the number of put and delete operations in a single batch write operation. If your request exceeds these limits, your request is rejected. If your table does not have sufficient provisioned throughput to serve this request, the unprocessed request items are returned in the response.

The following example checks the response to see if it has any unprocessed request items. If it does, it loops back and resends the `batchWriteItem` request with unprocessed items in the

request. If you followed the examples in this guide, you should already have created the `Forum` and `Thread` tables. You can also create these tables and upload sample data programmatically. For more information, see Creating example tables and uploading data using the AWS SDK for Java.

For step-by-step instructions for testing the following sample, see Java code examples.

Example

```
package com.amazonaws.codesamples.document;

import java.io.IOException;
import java.util.Arrays;
import java.util.HashSet;
import java.util.List;
import java.util.Map;

import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;
import com.amazonaws.services.dynamodbv2.document.BatchWriteItemOutcome;
import com.amazonaws.services.dynamodbv2.document.DynamoDB;
import com.amazonaws.services.dynamodbv2.document.Item;
import com.amazonaws.services.dynamodbv2.document.TableWriteItems;
import com.amazonaws.services.dynamodbv2.model.WriteRequest;

public class DocumentAPIBatchWrite {

    static AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();
    static DynamoDB dynamoDB = new DynamoDB(client);

    static String forumTableName = "Forum";
    static String threadTableName = "Thread";

    public static void main(String[] args) throws IOException {

        writeMultipleItemsBatchWrite();

    }

    private static void writeMultipleItemsBatchWrite() {
        try {

            // Add a new item to Forum
```

```
            TableWriteItems forumTableWriteItems = new
TableWriteItems(forumTableName) // Forum
                    .withItemsToPut(new Item().withPrimaryKey("Name", "Amazon
RDS").withNumber("Threads", 0));

            // Add a new item, and delete an existing item, from Thread
            // This table has a partition key and range key, so need to specify
            // both of them
            TableWriteItems threadTableWriteItems = new
TableWriteItems(threadTableName)
                    .withItemsToPut(
                            new Item().withPrimaryKey("ForumName", "Amazon RDS",
"Subject", "Amazon RDS Thread 1")
                                    .withString("Message", "ElastiCache Thread 1
message")
                                    .withStringSet("Tags", new
HashSet<String>(Arrays.asList("cache", "in-memory"))))
                    .withHashAndRangeKeysToDelete("ForumName", "Subject", "Amazon S3",
"S3 Thread 100");

            System.out.println("Making the request.");
            BatchWriteItemOutcome outcome =
dynamoDB.batchWriteItem(forumTableWriteItems, threadTableWriteItems);

            do {

                // Check for unprocessed keys which could happen if you exceed
                // provisioned throughput

                Map<String, List<WriteRequest>> unprocessedItems =
outcome.getUnprocessedItems();

                if (outcome.getUnprocessedItems().size() == 0) {
                    System.out.println("No unprocessed items found");
                } else {
                    System.out.println("Retrieving the unprocessed items");
                    outcome = dynamoDB.batchWriteItemUnprocessed(unprocessedItems);
                }

            } while (outcome.getUnprocessedItems().size() > 0);

        } catch (Exception e) {
            System.err.println("Failed to retrieve items: ");
            e.printStackTrace(System.err);
```

```
        }

    }

}
```

Example: Batch get operation using the AWS SDK for Java document API

The following Java code example uses the `batchGetItem` method to retrieve multiple items from the `Forum` and the `Thread` tables. The `BatchGetItemRequest` specifies the table names and a list of keys for each item to get. The example processes the response by printing the items retrieved.

> **Note**
>
> This code example assumes that you have already loaded data into DynamoDB for your account by following the instructions in the Creating tables and loading data for code examples in DynamoDB section.
> For step-by-step instructions to run the following example, see Java code examples.

Example

```
package com.amazonaws.codesamples.document;

import java.io.IOException;
import java.util.List;
import java.util.Map;

import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;
import com.amazonaws.services.dynamodbv2.document.BatchGetItemOutcome;
import com.amazonaws.services.dynamodbv2.document.DynamoDB;
import com.amazonaws.services.dynamodbv2.document.Item;
import com.amazonaws.services.dynamodbv2.document.TableKeysAndAttributes;
import com.amazonaws.services.dynamodbv2.model.KeysAndAttributes;

public class DocumentAPIBatchGet {
    static AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();
    static DynamoDB dynamoDB = new DynamoDB(client);
```

```
   static String forumTableName = "Forum";
   static String threadTableName = "Thread";

   public static void main(String[] args) throws IOException {
       retrieveMultipleItemsBatchGet();
   }

   private static void retrieveMultipleItemsBatchGet() {

       try {

           TableKeysAndAttributes forumTableKeysAndAttributes = new
TableKeysAndAttributes(forumTableName);
           // Add a partition key
           forumTableKeysAndAttributes.addHashOnlyPrimaryKeys("Name", "Amazon S3",
"Amazon DynamoDB");

           TableKeysAndAttributes threadTableKeysAndAttributes = new
TableKeysAndAttributes(threadTableName);
           // Add a partition key and a sort key
           threadTableKeysAndAttributes.addHashAndRangePrimaryKeys("ForumName",
"Subject", "Amazon DynamoDB",
                   "DynamoDB Thread 1", "Amazon DynamoDB", "DynamoDB Thread 2",
"Amazon S3", "S3 Thread 1");

           System.out.println("Making the request.");

           BatchGetItemOutcome outcome =
dynamoDB.batchGetItem(forumTableKeysAndAttributes,
                   threadTableKeysAndAttributes);

           Map<String, KeysAndAttributes> unprocessed = null;

           do {
               for (String tableName : outcome.getTableItems().keySet()) {
                   System.out.println("Items in table " + tableName);
                   List<Item> items = outcome.getTableItems().get(tableName);
                   for (Item item : items) {
                       System.out.println(item.toJSONPretty());
                   }
               }

               // Check for unprocessed keys which could happen if you exceed
```

```
                // provisioned
                // throughput or reach the limit on response size.
                unprocessed = outcome.getUnprocessedKeys();

                if (unprocessed.isEmpty()) {
                    System.out.println("No unprocessed keys found");
                } else {
                    System.out.println("Retrieving the unprocessed keys");
                    outcome = dynamoDB.batchGetItemUnprocessed(unprocessed);
                }

            } while (!unprocessed.isEmpty());

        } catch (Exception e) {
            System.err.println("Failed to retrieve items.");
            System.err.println(e.getMessage());
        }

    }

}
```

Example: Handling binary type attributes using the AWS SDK for Java document API

The following Java code example illustrates handling binary type attributes. The example adds an item to the `Reply` table. The item includes a binary type attribute (`ExtendedMessage`) that stores compressed data. The example then retrieves the item and prints all the attribute values. For illustration, the example uses the `GZIPOutputStream` class to compress a sample stream and assign it to the `ExtendedMessage` attribute. When the binary attribute is retrieved, it is decompressed using the `GZIPInputStream` class.

> **Note**
>
> The SDK for Java also provides an object persistence model, enabling you to map your client-side classes to DynamoDB tables. This approach can reduce the amount of code that you have to write. For more information, see Java 1.x: DynamoDBMapper.

If you followed the Creating tables and loading data for code examples in DynamoDB section, you should already have created the `Reply` table. You can also create this table programmatically. For more information, see Creating example tables and uploading data using the AWS SDK for Java.

For step-by-step instructions for testing the following sample, see Java code examples.

Example

```
package com.amazonaws.codesamples.document;

import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.IOException;
import java.nio.ByteBuffer;
import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.TimeZone;
import java.util.zip.GZIPInputStream;
import java.util.zip.GZIPOutputStream;

import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;
import com.amazonaws.services.dynamodbv2.document.DynamoDB;
import com.amazonaws.services.dynamodbv2.document.Item;
import com.amazonaws.services.dynamodbv2.document.Table;
import com.amazonaws.services.dynamodbv2.document.spec.GetItemSpec;

public class DocumentAPIItemBinaryExample {

    static AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();
    static DynamoDB dynamoDB = new DynamoDB(client);

    static String tableName = "Reply";
    static SimpleDateFormat dateFormatter = new SimpleDateFormat("yyyy-MM-
dd'T'HH:mm:ss.SSS'Z'");

    public static void main(String[] args) throws IOException {
        try {

            // Format the primary key values
            String threadId = "Amazon DynamoDB#DynamoDB Thread 2";
```

```
            dateFormatter.setTimeZone(TimeZone.getTimeZone("UTC"));
            String replyDateTime = dateFormatter.format(new Date());

            // Add a new reply with a binary attribute type
            createItem(threadId, replyDateTime);

            // Retrieve the reply with a binary attribute type
            retrieveItem(threadId, replyDateTime);

            // clean up by deleting the item
            deleteItem(threadId, replyDateTime);
        } catch (Exception e) {
            System.err.println("Error running the binary attribute type example: " +
e);
            e.printStackTrace(System.err);
        }
    }

    public static void createItem(String threadId, String replyDateTime) throws
IOException {

        Table table = dynamoDB.getTable(tableName);

        // Craft a long message
        String messageInput = "Long message to be compressed in a lengthy forum reply";

        // Compress the long message
        ByteBuffer compressedMessage = compressString(messageInput.toString());

        table.putItem(new Item().withPrimaryKey("Id",
threadId).withString("ReplyDateTime", replyDateTime)
                .withString("Message", "Long message
follows").withBinary("ExtendedMessage", compressedMessage)
                .withString("PostedBy", "User A"));
    }

    public static void retrieveItem(String threadId, String replyDateTime) throws
IOException {

        Table table = dynamoDB.getTable(tableName);

        GetItemSpec spec = new GetItemSpec().withPrimaryKey("Id", threadId,
"ReplyDateTime", replyDateTime)
                .withConsistentRead(true);
```

```
        Item item = table.getItem(spec);

        // Uncompress the reply message and print
        String uncompressed =
uncompressString(ByteBuffer.wrap(item.getBinary("ExtendedMessage")));

        System.out.println("Reply message:\n" + " Id: " + item.getString("Id") + "\n" +
" ReplyDateTime: "
                + item.getString("ReplyDateTime") + "\n" + " PostedBy: " +
item.getString("PostedBy") + "\n"
                + " Message: "
                + item.getString("Message") + "\n" + " ExtendedMessage (uncompressed):
" + uncompressed + "\n");
    }

    public static void deleteItem(String threadId, String replyDateTime) {

        Table table = dynamoDB.getTable(tableName);
        table.deleteItem("Id", threadId, "ReplyDateTime", replyDateTime);
    }

    private static ByteBuffer compressString(String input) throws IOException {
        // Compress the UTF-8 encoded String into a byte[]
        ByteArrayOutputStream baos = new ByteArrayOutputStream();
        GZIPOutputStream os = new GZIPOutputStream(baos);
        os.write(input.getBytes("UTF-8"));
        os.close();
        baos.close();
        byte[] compressedBytes = baos.toByteArray();

        // The following code writes the compressed bytes to a ByteBuffer.
        // A simpler way to do this is by simply calling
        // ByteBuffer.wrap(compressedBytes);
        // However, the longer form below shows the importance of resetting the
        // position of the buffer
        // back to the beginning of the buffer if you are writing bytes directly
        // to it, since the SDK
        // will consider only the bytes after the current position when sending
        // data to DynamoDB.
        // Using the "wrap" method automatically resets the position to zero.
        ByteBuffer buffer = ByteBuffer.allocate(compressedBytes.length);
        buffer.put(compressedBytes, 0, compressedBytes.length);
        buffer.position(0); // Important: reset the position of the ByteBuffer
```

```
                            // to the beginning
        return buffer;
    }

    private static String uncompressString(ByteBuffer input) throws IOException {
        byte[] bytes = input.array();
        ByteArrayInputStream bais = new ByteArrayInputStream(bytes);
        ByteArrayOutputStream baos = new ByteArrayOutputStream();
        GZIPInputStream is = new GZIPInputStream(bais);

        int chunkSize = 1024;
        byte[] buffer = new byte[chunkSize];
        int length = 0;
        while ((length = is.read(buffer, 0, chunkSize)) != -1) {
            baos.write(buffer, 0, length);
        }

        String result = new String(baos.toByteArray(), "UTF-8");

        is.close();
        baos.close();
        bais.close();

        return result;
    }
}
```

Working with items: .NET

You can use the AWS SDK for .NET low-level API to perform typical create, read, update, and delete (CRUD) operations on an item in a table. The following are the common steps that you follow to perform data CRUD operations using the .NET low-level API:

1. Create an instance of the `AmazonDynamoDBClient` class (the client).
2. Provide the operation-specific required parameters in a corresponding request object.

   For example, use the `PutItemRequest` request object when uploading an item and use the `GetItemRequest` request object when retrieving an existing item.

   You can use the request object to provide both the required and optional parameters.

3. Run the appropriate method provided by the client by passing in the request object that you created in the preceding step.

   The `AmazonDynamoDBClient` client provides `PutItem`, `GetItem`, `UpdateItem`, and `DeleteItem` methods for the CRUD operations.

Topics

- Putting an item
- Getting an item
- Updating an item
- Atomic counter
- Deleting an item
- Batch write: Putting and deleting multiple items
- Batch get: Getting multiple items
- Example: CRUD operations using the AWS SDK for .NET low-level API
- Example: Batch operations using the AWS SDK for .NET low-level API
- Example: Handling binary type attributes using the AWS SDK for .NET low-level API

Putting an item

The `PutItem` method uploads an item to a table. If the item exists, it replaces the entire item.

> **Note**
>
> Instead of replacing the entire item, if you want to update only specific attributes, you can use the `UpdateItem` method. For more information, see Updating an item.

The following are the steps to upload an item using the low-level .NET SDK API:

1. Create an instance of the `AmazonDynamoDBClient` class.
2. Provide the required parameters by creating an instance of the `PutItemRequest` class.

   To put an item, you must provide the table name and the item.

3. Run the `PutItem` method by providing the `PutItemRequest` object that you created in the preceding step.

The following C# example demonstrates the preceding steps. The example uploads an item to the `ProductCatalog` table.

Example

```
AmazonDynamoDBClient client = new AmazonDynamoDBClient();
string tableName = "ProductCatalog";

var request = new PutItemRequest
{
   TableName = tableName,
   Item = new Dictionary<string, AttributeValue>()
      {
          { "Id", new AttributeValue { N = "201" }},
          { "Title", new AttributeValue { S = "Book 201 Title" }},
          { "ISBN", new AttributeValue { S = "11-11-11-11" }},
          { "Price", new AttributeValue { S = "20.00" }},
          {
            "Authors",
            new AttributeValue
            { SS = new List<string>{"Author1", "Author2"}   }
          }
      }
};
client.PutItem(request);
```

In the preceding example, you upload a book item that has the `Id`, `Title`, `ISBN`, and `Authors` attributes. Note that `Id` is a numeric type attribute, and all other attributes are of the string type. Authors is a `String` set.

Specifying optional parameters

You can also provide optional parameters using the `PutItemRequest` object as shown in the following C# example. The example specifies the following optional parameters:

- `ExpressionAttributeNames`, `ExpressionAttributeValues`, and `ConditionExpression` specify that the item can be replaced only if the existing item has the ISBN attribute with a specific value.

- `ReturnValues` parameter to request the old item in the response.

Example

```
var request = new PutItemRequest
 {
   TableName = tableName,
   Item = new Dictionary<string, AttributeValue>()
               {
                   { "Id", new AttributeValue { N = "104" }},
                   { "Title", new AttributeValue { S = "Book 104  Title" }},
                   { "ISBN", new AttributeValue { S = "444-4444444444" }},
                   { "Authors",
                     new AttributeValue { SS = new List<string>{"Author3"}}}
               },
    // Optional parameters.
    ExpressionAttributeNames = new Dictionary<string,string>()
    {
        {"#I", "ISBN"}
    },
    ExpressionAttributeValues = new Dictionary<string, AttributeValue>()
    {
        {":isbn",new AttributeValue {S = "444-4444444444"}}
    },
    ConditionExpression = "#I = :isbn"

};
var response = client.PutItem(request);
```

For more information, see PutItem.

Getting an item

The `GetItem` method retrieves an item.

> **Note**
>
> To retrieve multiple items, you can use the `BatchGetItem` method. For more information, see Batch get: Getting multiple items.

The following are the steps to retrieve an existing item using the low-level AWS SDK for .NET API.

1. Create an instance of the `AmazonDynamoDBClient` class.
2. Provide the required parameters by creating an instance of the `GetItemRequest` class.

   To get an item, you must provide the table name and primary key of the item.
3. Run the `GetItem` method by providing the `GetItemRequest` object that you created in the preceding step.

The following C# example demonstrates the preceding steps. The example retrieves an item from the `ProductCatalog` table.

```
AmazonDynamoDBClient client = new AmazonDynamoDBClient();
string tableName = "ProductCatalog";

var request = new GetItemRequest
 {
   TableName = tableName,
   Key = new Dictionary<string,AttributeValue>() { { "Id", new AttributeValue { N =
 "202" } } },
 };
 var response = client.GetItem(request);

// Check the response.
var result = response.GetItemResult;
var attributeMap = result.Item; // Attribute list in the response.
```

Specifying optional parameters

You can also provide optional parameters using the `GetItemRequest` object, as shown in the following C# example. The sample specifies the following optional parameters:

- `ProjectionExpression` parameter to specify the attributes to retrieve.
- `ConsistentRead` parameter to perform a strongly consistent read. To learn more read consistency, see DynamoDB read consistency.

Example

```
AmazonDynamoDBClient client = new AmazonDynamoDBClient();
string tableName = "ProductCatalog";
```

```
var request = new GetItemRequest
 {
   TableName = tableName,
   Key = new Dictionary<string,AttributeValue>() { { "Id", new AttributeValue { N =
 "202" } } },
   // Optional parameters.
   ProjectionExpression = "Id, ISBN, Title, Authors",
   ConsistentRead = true
 };

 var response = client.GetItem(request);

// Check the response.
var result = response.GetItemResult;
var attributeMap = result.Item;
```

For more information, see GetItem.

Updating an item

The `UpdateItem` method updates an existing item if it is present. You can use the `UpdateItem` operation to update existing attribute values, add new attributes, or delete attributes from the existing collection. If the item that has the specified primary key is not found, it adds a new item.

The `UpdateItem` operation uses the following guidelines:

- If the item does not exist, `UpdateItem` adds a new item using the primary key that is specified in the input.
- If the item exists, `UpdateItem` applies the updates as follows:
  - Replaces the existing attribute values by the values in the update.
  - If the attribute that you provide in the input does not exist, it adds a new attribute to the item.
  - If the input attribute is null, it deletes the attribute, if it is present.
  - If you use `ADD` for the `Action`, you can add values to an existing set (string or number set), or mathematically add (use a positive number) or subtract (use a negative number) from the existing numeric attribute value.

> **Note**
>
> The `PutItem` operation also can perform an update. For more information, see Putting an item. For example, if you call `PutItem` to upload an item and the primary key exists, the `PutItem` operation replaces the entire item. If there are attributes in the existing item and those attributes are not specified in the input, the `PutItem` operation deletes those attributes. However, `UpdateItem` updates only the specified input attributes. Any other existing attributes of that item remain unchanged.

The following are the steps to update an existing item using the low-level .NET SDK API:

1. Create an instance of the `AmazonDynamoDBClient` class.
2. Provide the required parameters by creating an instance of the `UpdateItemRequest` class.

   This is the request object in which you describe all the updates, such as add attributes, update existing attributes, or delete attributes. To delete an existing attribute, specify the attribute name with null value.
3. Run the `UpdateItem` method by providing the `UpdateItemRequest` object that you created in the preceding step.

The following C# code example demonstrates the preceding steps. The example updates a book item in the `ProductCatalog` table. It adds a new author to the `Authors` collection, and deletes the existing `ISBN` attribute. It also reduces the price by one.

```
AmazonDynamoDBClient client = new AmazonDynamoDBClient();
string tableName = "ProductCatalog";

var request = new UpdateItemRequest
{
    TableName = tableName,
    Key = new Dictionary<string,AttributeValue>() { { "Id", new AttributeValue { N =
 "202" } } },
    ExpressionAttributeNames = new Dictionary<string,string>()
    {
        {"#A", "Authors"},
        {"#P", "Price"},
        {"#NA", "NewAttribute"},
```

```
        {"#I", "ISBN"}
    },
    ExpressionAttributeValues = new Dictionary<string, AttributeValue>()
    {
        {":auth",new AttributeValue { SS = {"Author YY","Author ZZ"}}},
        {":p",new AttributeValue {N = "1"}},
        {":newattr",new AttributeValue {S = "someValue"}},
    },

    // This expression does the following:
    // 1) Adds two new authors to the list
    // 2) Reduces the price
    // 3) Adds a new attribute to the item
    // 4) Removes the ISBN attribute from the item
    UpdateExpression = "ADD #A :auth SET #P = #P - :p, #NA = :newattr REMOVE #I"
};
var response = client.UpdateItem(request);
```

Specifying optional parameters

You can also provide optional parameters using the `UpdateItemRequest` object, as shown in the following C# example. It specifies the following optional parameters:

- `ExpressionAttributeValues` and `ConditionExpression` to specify that the price can be updated only if the existing price is 20.00.
- `ReturnValues` parameter to request the updated item in the response.

Example

```
AmazonDynamoDBClient client = new AmazonDynamoDBClient();
string tableName = "ProductCatalog";

var request = new UpdateItemRequest
{
    Key = new Dictionary<string,AttributeValue>() { { "Id", new AttributeValue { N =
 "202" } } },

    // Update price only if the current price is 20.00.
    ExpressionAttributeNames = new Dictionary<string,string>()
    {
        {"#P", "Price"}
    },
```

```
    ExpressionAttributeValues = new Dictionary<string, AttributeValue>()
    {
        {":newprice",new AttributeValue {N = "22"}},
        {":currprice",new AttributeValue {N = "20"}}
    },
    UpdateExpression = "SET #P = :newprice",
    ConditionExpression = "#P = :currprice",
    TableName = tableName,
    ReturnValues = "ALL_NEW" // Return all the attributes of the updated item.
};

var response = client.UpdateItem(request);
```

For more information, see UpdateItem.

Atomic counter

You can use `updateItem` to implement an atomic counter, where you increment or decrement the value of an existing attribute without interfering with other write requests. To update an atomic counter, use `updateItem` with an attribute of type `Number` in the `UpdateExpression` parameter, and `ADD` as the `Action`.

The following example demonstrates this, incrementing the `Quantity` attribute by one.

```
AmazonDynamoDBClient client = new AmazonDynamoDBClient();
string tableName = "ProductCatalog";

var request = new UpdateItemRequest
{
    Key = new Dictionary<string, AttributeValue>() { { "Id", new AttributeValue { N =
 "121" } } },
    ExpressionAttributeNames = new Dictionary<string, string>()
    {
        {"#Q", "Quantity"}
    },
    ExpressionAttributeValues = new Dictionary<string, AttributeValue>()
    {
        {":incr",new AttributeValue {N = "1"}}
    },
    UpdateExpression = "SET #Q = #Q + :incr",
    TableName = tableName
};
```

```
var response = client.UpdateItem(request);
```

Deleting an item

The `DeleteItem` method deletes an item from a table.

The following are the steps to delete an item using the low-level .NET SDK API.

1. Create an instance of the `AmazonDynamoDBClient` class.
2. Provide the required parameters by creating an instance of the `DeleteItemRequest` class.

   To delete an item, the table name and item's primary key are required.
3. Run the `DeleteItem` method by providing the `DeleteItemRequest` object that you created in the preceding step.

Example

```
AmazonDynamoDBClient client = new AmazonDynamoDBClient();
string tableName = "ProductCatalog";

var request = new DeleteItemRequest
{
    TableName = tableName,
    Key = new Dictionary<string,AttributeValue>() { { "Id", new AttributeValue { N =
 "201" } } },
};

var response = client.DeleteItem(request);
```

Specifying optional parameters

You can also provide optional parameters using the `DeleteItemRequest` object as shown in the following C# code example. It specifies the following optional parameters:

- `ExpressionAttributeValues` and `ConditionExpression` to specify that the book item can be deleted only if it is no longer in publication (the InPublication attribute value is false).
- `ReturnValues` parameter to request the deleted item in the response.

Example

```
var request = new DeleteItemRequest
{
    TableName = tableName,
    Key = new Dictionary<string,AttributeValue>() { { "Id", new AttributeValue { N =
 "201" } } },

    // Optional parameters.
    ReturnValues = "ALL_OLD",
    ExpressionAttributeNames = new Dictionary<string, string>()
    {
        {"#IP", "InPublication"}
    },
    ExpressionAttributeValues = new Dictionary<string, AttributeValue>()
    {
        {":inpub",new AttributeValue {BOOL = false}}
    },
    ConditionExpression = "#IP = :inpub"
};

var response = client.DeleteItem(request);
```

For more information, see DeleteItem.

Batch write: Putting and deleting multiple items

Batch write refers to putting and deleting multiple items in a batch. The `BatchWriteItem` method enables you to put and delete multiple items from one or more tables in a single call. The following are the steps to retrieve multiple items using the low-level .NET SDK API.

1. Create an instance of the `AmazonDynamoDBClient` class.
2. Describe all the put and delete operations by creating an instance of the `BatchWriteItemRequest` class.
3. Run the `BatchWriteItem` method by providing the `BatchWriteItemRequest` object that you created in the preceding step.
4. Process the response. You should check if there were any unprocessed request items returned in the response. This could happen if you reach the provisioned throughput quota or some other transient error. Also, DynamoDB limits the request size and the number of operations

you can specify in a request. If you exceed these limits, DynamoDB rejects the request. For more information, see BatchWriteItem.

The following C# code example demonstrates the preceding steps. The example creates a `BatchWriteItemRequest` to perform the following write operations:

- Put an item in `Forum` table.
- Put and delete an item from `Thread` table.

The code runs `BatchWriteItem` to perform a batch operation.

```
AmazonDynamoDBClient client = new AmazonDynamoDBClient();

string table1Name = "Forum";
string table2Name = "Thread";

var request = new BatchWriteItemRequest
 {
   RequestItems = new Dictionary<string, List<WriteRequest>>
    {
      {
        table1Name, new List<WriteRequest>
        {
          new WriteRequest
          {
             PutRequest = new PutRequest
             {
                Item = new Dictionary<string,AttributeValue>
                {
                  { "Name", new AttributeValue { S = "Amazon S3 forum" } },
                  { "Threads", new AttributeValue { N = "0" }}
                }
             }
          }
        }
      } ,
      {
        table2Name, new List<WriteRequest>
        {
          new WriteRequest
          {
```

```
            PutRequest = new PutRequest
            {
               Item = new Dictionary<string,AttributeValue>
               {
                 { "ForumName", new AttributeValue { S = "Amazon S3 forum" } },
                 { "Subject", new AttributeValue { S = "My sample question" } },
                 { "Message", new AttributeValue { S = "Message Text." } },
                 { "KeywordTags", new AttributeValue { SS = new List<string> { "Amazon
 S3", "Bucket" }  } }
               }
            }
         },
         new WriteRequest
         {
            DeleteRequest = new DeleteRequest
            {
               Key = new Dictionary<string,AttributeValue>()
               {
                  { "ForumName", new AttributeValue { S = "Some forum name" } },
                  { "Subject", new AttributeValue { S = "Some subject" } }
               }
            }
         }
       }
     }
   }
 };
response = client.BatchWriteItem(request);
```

For a working example, see Example: Batch operations using the AWS SDK for .NET low-level API.

Batch get: Getting multiple items

The `BatchGetItem` method enables you to retrieve multiple items from one or more tables.

> **Note**
>
> To retrieve a single item, you can use the `GetItem` method.

The following are the steps to retrieve multiple items using the low-level AWS SDK for .NET API.

1. Create an instance of the `AmazonDynamoDBClient` class.

2. Provide the required parameters by creating an instance of the `BatchGetItemRequest` class.

   To retrieve multiple items, the table name and a list of primary key values are required.
3. Run the `BatchGetItem` method by providing the `BatchGetItemRequest` object that you created in the preceding step.
4. Process the response. You should check if there were any unprocessed keys, which could happen if you reach the provisioned throughput quota or some other transient error.

The following C# code example demonstrates the preceding steps. The example retrieves items from two tables, `Forum` and `Thread`. The request specifies two items in the `Forum` and three items in the `Thread` table. The response includes items from both of the tables. The code shows how you can process the response.

```
AmazonDynamoDBClient client = new AmazonDynamoDBClient();

string table1Name = "Forum";
string table2Name = "Thread";

var request = new BatchGetItemRequest
{
  RequestItems = new Dictionary<string, KeysAndAttributes>()
  {
    { table1Name,
      new KeysAndAttributes
      {
        Keys = new List<Dictionary<string, AttributeValue>>()
        {
          new Dictionary<string, AttributeValue>()
          {
            { "Name", new AttributeValue { S = "DynamoDB" } }
          },
          new Dictionary<string, AttributeValue>()
          {
            { "Name", new AttributeValue { S = "Amazon S3" } }
          }
        }
      }
    },
    {
      table2Name,
```

```
      new KeysAndAttributes
      {
        Keys = new List<Dictionary<string, AttributeValue>>()
        {
          new Dictionary<string, AttributeValue>()
          {
            { "ForumName", new AttributeValue { S = "DynamoDB" } },
            { "Subject", new AttributeValue { S = "DynamoDB Thread 1" } }
          },
          new Dictionary<string, AttributeValue>()
          {
            { "ForumName", new AttributeValue { S = "DynamoDB" } },
            { "Subject", new AttributeValue { S = "DynamoDB Thread 2" } }
          },
          new Dictionary<string, AttributeValue>()
          {
            { "ForumName", new AttributeValue { S = "Amazon S3" } },
            { "Subject", new AttributeValue { S = "Amazon S3 Thread 1" } }
          }
        }
      }
    }
  }
};

var response = client.BatchGetItem(request);

// Check the response.
var result = response.BatchGetItemResult;
var responses = result.Responses; // The attribute list in the response.

var table1Results = responses[table1Name];
Console.WriteLine("Items in table {0}" + table1Name);
foreach (var item1 in table1Results.Items)
{
  PrintItem(item1);
}

var table2Results = responses[table2Name];
Console.WriteLine("Items in table {1}" + table2Name);
foreach (var item2 in table2Results.Items)
{
  PrintItem(item2);
}
```

```
// Any unprocessed keys? could happen if you exceed ProvisionedThroughput or some other
 error.
Dictionary<string, KeysAndAttributes> unprocessedKeys = result.UnprocessedKeys;
foreach (KeyValuePair<string, KeysAndAttributes> pair in unprocessedKeys)
{
    Console.WriteLine(pair.Key, pair.Value);
}
```

Specifying optional parameters

You can also provide optional parameters using the `BatchGetItemRequest` object as shown in the following C# code example. The example retrieves two items from the `Forum` table. It specifies the following optional parameter:

- `ProjectionExpression` parameter to specify the attributes to retrieve.

Example

```
AmazonDynamoDBClient client = new AmazonDynamoDBClient();

string table1Name = "Forum";

var request = new BatchGetItemRequest
{
  RequestItems = new Dictionary<string, KeysAndAttributes>()
  {
    { table1Name,
      new KeysAndAttributes
      {
        Keys = new List<Dictionary<string, AttributeValue>>()
        {
          new Dictionary<string, AttributeValue>()
          {
            { "Name", new AttributeValue { S = "DynamoDB" } }
          },
          new Dictionary<string, AttributeValue>()
          {
            { "Name", new AttributeValue { S = "Amazon S3" } }
          }
        }
      },
      // Optional - name of an attribute to retrieve.
```

```
      ProjectionExpression = "Title"
    }
  }
};

var response = client.BatchGetItem(request);
```

For more information, see BatchGetItem.

Example: CRUD operations using the AWS SDK for .NET low-level API

The following C# code example illustrates CRUD operations on an Amazon DynamoDB item. The example adds an item to the `ProductCatalog` table, retrieves it, performs various updates, and finally deletes the item. If you haven't created this table, you can also create it programmatically. For more information, see Creating example tables and uploading data using the AWS SDK for .NET.

For step-by-step instructions for testing the following sample, see .NET code examples.

Example

```
using System;
using System.Collections.Generic;
using Amazon.DynamoDBv2;
using Amazon.DynamoDBv2.Model;
using Amazon.Runtime;
using Amazon.SecurityToken;

namespace com.amazonaws.codesamples
{
    class LowLevelItemCRUDExample
    {
        private static string tableName = "ProductCatalog";
        private static AmazonDynamoDBClient client = new AmazonDynamoDBClient();

        static void Main(string[] args)
        {
            try
            {
                CreateItem();
                RetrieveItem();

                // Perform various updates.
```

```
            UpdateMultipleAttributes();
            UpdateExistingAttributeConditionally();

            // Delete item.
            DeleteItem();
            Console.WriteLine("To continue, press Enter");
            Console.ReadLine();
        }
        catch (Exception e)
        {
            Console.WriteLine(e.Message);
            Console.WriteLine("To continue, press Enter");
            Console.ReadLine();
        }
    }

    private static void CreateItem()
    {
        var request = new PutItemRequest
        {
            TableName = tableName,
            Item = new Dictionary<string, AttributeValue>()
        {
            { "Id", new AttributeValue {
                  N = "1000"
              }},
            { "Title", new AttributeValue {
                  S = "Book 201 Title"
              }},
            { "ISBN", new AttributeValue {
                  S = "11-11-11-11"
              }},
            { "Authors", new AttributeValue {
                  SS = new List<string>{"Author1", "Author2" }
              }},
            { "Price", new AttributeValue {
                  N = "20.00"
              }},
            { "Dimensions", new AttributeValue {
                  S = "8.5x11.0x.75"
              }},
            { "InPublication", new AttributeValue {
                  BOOL = false
              } }
```

```
            }
            };
            client.PutItem(request);
        }

        private static void RetrieveItem()
        {
            var request = new GetItemRequest
            {
                TableName = tableName,
                Key = new Dictionary<string, AttributeValue>()
            {
                { "Id", new AttributeValue {
                      N = "1000"
                  } }
            },
                ProjectionExpression = "Id, ISBN, Title, Authors",
                ConsistentRead = true
            };
            var response = client.GetItem(request);

            // Check the response.
            var attributeList = response.Item; // attribute list in the response.
            Console.WriteLine("\nPrinting item after retrieving it ............");
            PrintItem(attributeList);
        }

        private static void UpdateMultipleAttributes()
        {
            var request = new UpdateItemRequest
            {
                Key = new Dictionary<string, AttributeValue>()
            {
                { "Id", new AttributeValue {
                      N = "1000"
                  } }
            },
                // Perform the following updates:
                // 1) Add two new authors to the list
                // 1) Set a new attribute
                // 2) Remove the ISBN attribute
                ExpressionAttributeNames = new Dictionary<string, string>()
            {
                {"#A","Authors"},
```

```
                {"#NA","NewAttribute"},
                {"#I","ISBN"}
            },
                ExpressionAttributeValues = new Dictionary<string, AttributeValue>()
            {
                {":auth",new AttributeValue {
                     SS = {"Author YY", "Author ZZ"}
                 }},
                {":new",new AttributeValue {
                     S = "New Value"
                 }}
            },

                UpdateExpression = "ADD #A :auth SET #NA = :new REMOVE #I",

                TableName = tableName,
                ReturnValues = "ALL_NEW" // Give me all attributes of the updated item.
            };
            var response = client.UpdateItem(request);

            // Check the response.
            var attributeList = response.Attributes; // attribute list in the response.
                                                     // print attributeList.
            Console.WriteLine("\nPrinting item after multiple attribute
update .............");
            PrintItem(attributeList);
        }

        private static void UpdateExistingAttributeConditionally()
        {
            var request = new UpdateItemRequest
            {
                Key = new Dictionary<string, AttributeValue>()
            {
                { "Id", new AttributeValue {
                      N = "1000"
                  } }
            },
                ExpressionAttributeNames = new Dictionary<string, string>()
            {
                {"#P", "Price"}
            },
                ExpressionAttributeValues = new Dictionary<string, AttributeValue>()
            {
```

```
                {":newprice",new AttributeValue {
                     N = "22.00"
                 }},
                {":currprice",new AttributeValue {
                     N = "20.00"
                 }}
            },
                // This updates price only if current price is 20.00.
                UpdateExpression = "SET #P = :newprice",
                ConditionExpression = "#P = :currprice",

                TableName = tableName,
                ReturnValues = "ALL_NEW" // Give me all attributes of the updated item.
            };
            var response = client.UpdateItem(request);

            // Check the response.
            var attributeList = response.Attributes; // attribute list in the response.
            Console.WriteLine("\nPrinting item after updating price value
conditionally ............");
            PrintItem(attributeList);
        }

        private static void DeleteItem()
        {
            var request = new DeleteItemRequest
            {
                TableName = tableName,
                Key = new Dictionary<string, AttributeValue>()
            {
                { "Id", new AttributeValue {
                      N = "1000"
                  } }
            },

                // Return the entire item as it appeared before the update.
                ReturnValues = "ALL_OLD",
                ExpressionAttributeNames = new Dictionary<string, string>()
            {
                {"#IP", "InPublication"}
            },
                ExpressionAttributeValues = new Dictionary<string, AttributeValue>()
            {
                {":inpub",new AttributeValue {
```

```
                    BOOL = false
                }}
            },
                ConditionExpression = "#IP = :inpub"
            };

            var response = client.DeleteItem(request);

            // Check the response.
            var attributeList = response.Attributes; // Attribute list in the response.
                                                     // Print item.
            Console.WriteLine("\nPrinting item that was just deleted ............");
            PrintItem(attributeList);
        }

        private static void PrintItem(Dictionary<string, AttributeValue> attributeList)
        {
            foreach (KeyValuePair<string, AttributeValue> kvp in attributeList)
            {
                string attributeName = kvp.Key;
                AttributeValue value = kvp.Value;

                Console.WriteLine(
                    attributeName + " " +
                    (value.S == null ? "" : "S=[" + value.S + "]") +
                    (value.N == null ? "" : "N=[" + value.N + "]") +
                    (value.SS == null ? "" : "SS=[" + string.Join(",",
 value.SS.ToArray()) + "]") +
                    (value.NS == null ? "" : "NS=[" + string.Join(",",
 value.NS.ToArray()) + "]")
                    );
            }
            Console.WriteLine("************************************************");
        }
    }
}
```

Example: Batch operations using the AWS SDK for .NET low-level API

Topics

- Example: Batch write operation using the AWS SDK for .NET low-level API
- Example: Batch get operation using the AWS SDK for .NET low-level API

This section provides examples of batch operations, *batch write* and *batch get*, that Amazon DynamoDB supports.

Example: Batch write operation using the AWS SDK for .NET low-level API

The following C# code example uses the `BatchWriteItem` method to perform the following put and delete operations:

- Put one item in the `Forum` table.
- Put one item and delete one item from the `Thread` table.

You can specify any number of put and delete requests against one or more tables when creating your batch write request. However, DynamoDB `BatchWriteItem` limits the size of a batch write request and the number of put and delete operations in a single batch write operation. For more information, see BatchWriteItem. If your request exceeds these limits, your request is rejected. If your table does not have sufficient provisioned throughput to serve this request, the unprocessed request items are returned in the response.

The following example checks the response to see if it has any unprocessed request items. If it does, it loops back and resends the `BatchWriteItem` request with unprocessed items in the request. You can also create these sample tables and upload sample data programmatically. For more information, see Creating example tables and uploading data using the AWS SDK for .NET.

For step-by-step instructions for testing the following sample, see .NET code examples.

Example

```
using System;
using System.Collections.Generic;
using Amazon.DynamoDBv2;
using Amazon.DynamoDBv2.Model;
using Amazon.Runtime;

namespace com.amazonaws.codesamples
{
    class LowLevelBatchWrite
    {
        private static string table1Name = "Forum";
        private static string table2Name = "Thread";
        private static AmazonDynamoDBClient client = new AmazonDynamoDBClient();
```

```
        static void Main(string[] args)
        {
            try
            {
                TestBatchWrite();
            }
            catch (AmazonServiceException e) { Console.WriteLine(e.Message); }
            catch (Exception e) { Console.WriteLine(e.Message); }

            Console.WriteLine("To continue, press Enter");
            Console.ReadLine();
        }

        private static void TestBatchWrite()
        {
            var request = new BatchWriteItemRequest
            {
                ReturnConsumedCapacity = "TOTAL",
                RequestItems = new Dictionary<string, List<WriteRequest>>
            {
                {
                    table1Name, new List<WriteRequest>
                    {
                        new WriteRequest
                        {
                            PutRequest = new PutRequest
                            {
                                Item = new Dictionary<string, AttributeValue>
                                {
                                    { "Name", new AttributeValue {
                                          S = "S3 forum"
                                      } },
                                    { "Threads", new AttributeValue {
                                          N = "0"
                                      }}
                                }
                            }
                        }
                    }
                },
                {
                    table2Name, new List<WriteRequest>
                    {
                        new WriteRequest
```

```
                        {
                            PutRequest = new PutRequest
                            {
                                Item = new Dictionary<string, AttributeValue>
                                {
                                    { "ForumName", new AttributeValue {
                                          S = "S3 forum"
                                      } },
                                    { "Subject", new AttributeValue {
                                          S = "My sample question"
                                      } },
                                    { "Message", new AttributeValue {
                                          S = "Message Text."
                                      } },
                                    { "KeywordTags", new AttributeValue {
                                          SS = new List<string> { "S3", "Bucket" }
                                      } }
                                }
                            }
                        },
                        new WriteRequest
                        {
                            // For the operation to delete an item, if you provide a
primary key value
                            // that does not exist in the table, there is no error, it
is just a no-op.
                            DeleteRequest = new DeleteRequest
                            {
                                Key = new Dictionary<string, AttributeValue>()
                                {
                                    { "ForumName",  new AttributeValue {
                                          S = "Some partition key value"
                                      } },
                                    { "Subject", new AttributeValue {
                                          S = "Some sort key value"
                                      } }
                                }
                            }
                        }
                    }
                }
            }
            };
```

```
            CallBatchWriteTillCompletion(request);
        }

        private static void CallBatchWriteTillCompletion(BatchWriteItemRequest request)
        {
            BatchWriteItemResponse response;

            int callCount = 0;
            do
            {
                Console.WriteLine("Making request");
                response = client.BatchWriteItem(request);
                callCount++;

                // Check the response.

                var tableConsumedCapacities = response.ConsumedCapacity;
                var unprocessed = response.UnprocessedItems;

                Console.WriteLine("Per-table consumed capacity");
                foreach (var tableConsumedCapacity in tableConsumedCapacities)
                {
                    Console.WriteLine("{0} - {1}", tableConsumedCapacity.TableName,
 tableConsumedCapacity.CapacityUnits);
                }

                Console.WriteLine("Unprocessed");
                foreach (var unp in unprocessed)
                {
                    Console.WriteLine("{0} - {1}", unp.Key, unp.Value.Count);
                }
                Console.WriteLine();

                // For the next iteration, the request will have unprocessed items.
                request.RequestItems = unprocessed;
            } while (response.UnprocessedItems.Count > 0);

            Console.WriteLine("Total # of batch write API calls made: {0}", callCount);
        }
    }
}
```

Example: Batch get operation using the AWS SDK for .NET low-level API

The following C# code example uses the `BatchGetItem` method to retrieve multiple items from the `Forum` and the `Thread` tables in Amazon DynamoDB. The `BatchGetItemRequest` specifies the table names and a list of primary keys for each table. The example processes the response by printing the items retrieved.

For step-by-step instructions for testing the following sample, see .NET code examples.

Example

```
using System;
using System.Collections.Generic;
using Amazon.DynamoDBv2;
using Amazon.DynamoDBv2.Model;
using Amazon.Runtime;

namespace com.amazonaws.codesamples
{
    class LowLevelBatchGet
    {
        private static string table1Name = "Forum";
        private static string table2Name = "Thread";
        private static AmazonDynamoDBClient client = new AmazonDynamoDBClient();

        static void Main(string[] args)
        {
            try
            {
                RetrieveMultipleItemsBatchGet();

                Console.WriteLine("To continue, press Enter");
                Console.ReadLine();
            }
            catch (AmazonServiceException e) { Console.WriteLine(e.Message); }
            catch (Exception e) { Console.WriteLine(e.Message); }
        }

        private static void RetrieveMultipleItemsBatchGet()
        {
            var request = new BatchGetItemRequest
            {
                RequestItems = new Dictionary<string, KeysAndAttributes>()
```

```
            {
                { table1Name,
                  new KeysAndAttributes
                  {
                      Keys = new List<Dictionary<string, AttributeValue> >()
                      {
                          new Dictionary<string, AttributeValue>()
                          {
                              { "Name", new AttributeValue {
                            S = "Amazon DynamoDB"
                          } }
                            },
                          new Dictionary<string, AttributeValue>()
                          {
                              { "Name", new AttributeValue {
                            S = "Amazon S3"
                          } }
                            }
                      }
                  }},
                {
                    table2Name,
                    new KeysAndAttributes
                    {
                        Keys = new List<Dictionary<string, AttributeValue> >()
                        {
                            new Dictionary<string, AttributeValue>()
                            {
                                { "ForumName", new AttributeValue {
                                      S = "Amazon DynamoDB"
                                  } },
                                { "Subject", new AttributeValue {
                                      S = "DynamoDB Thread 1"
                                  } }
                            },
                            new Dictionary<string, AttributeValue>()
                            {
                                { "ForumName", new AttributeValue {
                                      S = "Amazon DynamoDB"
                                  } },
                                { "Subject", new AttributeValue {
                                      S = "DynamoDB Thread 2"
                                  } }
                            },
```

```
                            new Dictionary<string, AttributeValue>()
                            {
                                { "ForumName", new AttributeValue {
                                      S = "Amazon S3"
                                  } },
                                { "Subject", new AttributeValue {
                                      S = "S3 Thread 1"
                                  } }
                            }
                        }
                    }
                }
            }
            };

            BatchGetItemResponse response;
            do
            {
                Console.WriteLine("Making request");
                response = client.BatchGetItem(request);

                // Check the response.
                var responses = response.Responses; // Attribute list in the response.

                foreach (var tableResponse in responses)
                {
                    var tableResults = tableResponse.Value;
                    Console.WriteLine("Items retrieved from table {0}",
tableResponse.Key);
                    foreach (var item1 in tableResults)
                    {
                        PrintItem(item1);
                    }
                }

                // Any unprocessed keys? could happen if you exceed
ProvisionedThroughput or some other error.
                Dictionary<string, KeysAndAttributes> unprocessedKeys =
response.UnprocessedKeys;
                foreach (var unprocessedTableKeys in unprocessedKeys)
                {
                    // Print table name.
                    Console.WriteLine(unprocessedTableKeys.Key);
                    // Print unprocessed primary keys.
```

```
                    foreach (var key in unprocessedTableKeys.Value.Keys)
                    {
                        PrintItem(key);
                    }
                }

                request.RequestItems = unprocessedKeys;
            } while (response.UnprocessedKeys.Count > 0);
        }

        private static void PrintItem(Dictionary<string, AttributeValue> attributeList)
        {
            foreach (KeyValuePair<string, AttributeValue> kvp in attributeList)
            {
                string attributeName = kvp.Key;
                AttributeValue value = kvp.Value;

                Console.WriteLine(
                    attributeName + " " +
                    (value.S == null ? "" : "S=[" + value.S + "]") +
                    (value.N == null ? "" : "N=[" + value.N + "]") +
                    (value.SS == null ? "" : "SS=[" + string.Join(",",
 value.SS.ToArray()) + "]") +
                    (value.NS == null ? "" : "NS=[" + string.Join(",",
 value.NS.ToArray()) + "]")
                    );
            }
            Console.WriteLine("************************************************");
        }
    }
}
```

Example: Handling binary type attributes using the AWS SDK for .NET low-level API

The following C# code example illustrates the handling of binary type attributes. The example adds an item to the `Reply` table. The item includes a binary type attribute (`ExtendedMessage`) that stores compressed data. The example then retrieves the item and prints all the attribute values. For illustration, the example uses the `GZipStream` class to compress a sample stream and assigns it to the `ExtendedMessage` attribute, and decompresses it when printing the attribute value.

For step-by-step instructions for testing the following example, see .NET code examples.

Example

```
using System;
using System.Collections.Generic;
using System.IO;
using System.IO.Compression;
using Amazon.DynamoDBv2;
using Amazon.DynamoDBv2.Model;
using Amazon.Runtime;

namespace com.amazonaws.codesamples
{
    class LowLevelItemBinaryExample
    {
        private static string tableName = "Reply";
        private static AmazonDynamoDBClient client = new AmazonDynamoDBClient();

        static void Main(string[] args)
        {
            // Reply table primary key.
            string replyIdPartitionKey = "Amazon DynamoDB#DynamoDB Thread 1";
            string replyDateTimeSortKey = Convert.ToString(DateTime.UtcNow);

            try
            {
                CreateItem(replyIdPartitionKey, replyDateTimeSortKey);
                RetrieveItem(replyIdPartitionKey, replyDateTimeSortKey);
                // Delete item.
                DeleteItem(replyIdPartitionKey, replyDateTimeSortKey);
                Console.WriteLine("To continue, press Enter");
                Console.ReadLine();
            }
            catch (AmazonDynamoDBException e) { Console.WriteLine(e.Message); }
            catch (AmazonServiceException e) { Console.WriteLine(e.Message); }
            catch (Exception e) { Console.WriteLine(e.Message); }
        }

        private static void CreateItem(string partitionKey, string sortKey)
        {
            MemoryStream compressedMessage = ToGzipMemoryStream("Some long extended
 message to compress.");
            var request = new PutItemRequest
            {
                TableName = tableName,
```

```
            Item = new Dictionary<string, AttributeValue>()
        {
            { "Id", new AttributeValue {
                  S = partitionKey
              }},
            { "ReplyDateTime", new AttributeValue {
                  S = sortKey
              }},
            { "Subject", new AttributeValue {
                  S = "Binary type "
              }},
            { "Message", new AttributeValue {
                  S = "Some message about the binary type"
              }},
            { "ExtendedMessage", new AttributeValue {
                  B = compressedMessage
              }}
        }
        };
        client.PutItem(request);
    }

    private static void RetrieveItem(string partitionKey, string sortKey)
    {
        var request = new GetItemRequest
        {
            TableName = tableName,
            Key = new Dictionary<string, AttributeValue>()
        {
            { "Id", new AttributeValue {
                  S = partitionKey
              } },
            { "ReplyDateTime", new AttributeValue {
                  S = sortKey
              } }
        },
            ConsistentRead = true
        };
        var response = client.GetItem(request);

        // Check the response.
        var attributeList = response.Item; // attribute list in the response.
        Console.WriteLine("\nPrinting item after retrieving it ............");
```

```
            PrintItem(attributeList);
        }

        private static void DeleteItem(string partitionKey, string sortKey)
        {
            var request = new DeleteItemRequest
            {
                TableName = tableName,
                Key = new Dictionary<string, AttributeValue>()
            {
                { "Id", new AttributeValue {
                      S = partitionKey
                  } },
                { "ReplyDateTime", new AttributeValue {
                      S = sortKey
                  } }
            }
            };
            var response = client.DeleteItem(request);
        }

        private static void PrintItem(Dictionary<string, AttributeValue> attributeList)
        {
            foreach (KeyValuePair<string, AttributeValue> kvp in attributeList)
            {
                string attributeName = kvp.Key;
                AttributeValue value = kvp.Value;

                Console.WriteLine(
                    attributeName + " " +
                    (value.S == null ? "" : "S=[" + value.S + "]") +
                    (value.N == null ? "" : "N=[" + value.N + "]") +
                    (value.SS == null ? "" : "SS=[" + string.Join(",",
value.SS.ToArray()) + "]") +
                    (value.NS == null ? "" : "NS=[" + string.Join(",",
value.NS.ToArray()) + "]") +
                    (value.B == null ? "" : "B=[" + FromGzipMemoryStream(value.B) +
"]")
                    );
            }
            Console.WriteLine("*********************************************");
        }

        private static MemoryStream ToGzipMemoryStream(string value)
```

```
        {
            MemoryStream output = new MemoryStream();
            using (GZipStream zipStream = new GZipStream(output,
 CompressionMode.Compress, true))
            using (StreamWriter writer = new StreamWriter(zipStream))
            {
                writer.Write(value);
            }
            return output;
        }

        private static string FromGzipMemoryStream(MemoryStream stream)
        {
            using (GZipStream zipStream = new GZipStream(stream,
 CompressionMode.Decompress))
            using (StreamReader reader = new StreamReader(zipStream))
            {
                return reader.ReadToEnd();
            }
        }
    }
}
```

Improving data access with secondary indexes in DynamoDB

Amazon DynamoDB provides fast access to items in a table by specifying primary key values. However, many applications might benefit from having one or more secondary (or alternate) keys available, to allow efficient access to data with attributes other than the primary key. To address this, you can create one or more secondary indexes on a table and issue `Query` or `Scan` requests against these indexes.

A *secondary index* is a data structure that contains a subset of attributes from a table, along with an alternate key to support `Query` operations. You can retrieve data from the index using a `Query`, in much the same way as you use `Query` with a table. A table can have multiple secondary indexes, which give your applications access to many different query patterns.

Note

You can also `Scan` an index, in much the same way as you would `Scan` a table.

Cross-account access for secondary index scan operations is currently not supported with resource-based policies.

Every secondary index is associated with exactly one table, from which it obtains its data. This is called the *base table* for the index. When you create an index, you define an alternate key for the index (partition key and sort key). You also define the attributes that you want to be *projected*, or copied, from the base table into the index. DynamoDB copies these attributes into the index, along with the primary key attributes from the base table. You can then query or scan the index just as you would query or scan a table.

Every secondary index is automatically maintained by DynamoDB. When you add, modify, or delete items in the base table, any indexes on that table are also updated to reflect these changes.

DynamoDB supports two types of secondary indexes:

- **Global secondary index** — An index with a partition key and a sort key that can be different from those on the base table. A global secondary index is considered "global" because queries on the index can span all of the data in the base table, across all partitions. A global secondary index is stored in its own partition space away from the base table and scales separately from the base table.
- **Local secondary index** — An index that has the same partition key as the base table, but a different sort key. A local secondary index is "local" in the sense that every partition of a local secondary index is scoped to a base table partition that has the same partition key value.

For a comparison of global secondary indexes and local secondary indexes, see this video.

Making the right choice between GSI and LSI

Topics

- Using Global Secondary Indexes in DynamoDB
- Local secondary indexes

You should consider your application's requirements when you determine which type of index to use. The following table shows the main differences between a global secondary index and a local secondary index.

Characteristic	Global secondary index	Local secondary index
Key Schema	The primary key of a global secondary index can be either simple (partition key) or composite (partition key and sort key).	The primary key of a local secondary index must be composite (partition key and sort key).
Key Attributes	The index partition key and sort key (if present) can be any base table attributes of type string, number, or binary.	The partition key of the index is the same attribute as the partition key of the base table. The sort key can be any base table attribute of type string, number, or binary.
Size Restrictions Per Partition Key Value	There are no size restrictions for global secondary indexes.	For each partition key value, the total size of all indexed items must be 10 GB or less.
Online Index Operations	Global secondary indexes can be created at the same time that you create a table. You can also add a new global secondary index to an existing table, or delete an existing global secondary index. For more information, see Managing Global Secondary Indexes in DynamoDB.	Local secondary indexes are created at the same time that you create a table. You cannot add a local secondary index to an existing table, nor can you delete any local secondary indexes that currently exist.
Queries and Partitions	A global secondary index lets you query over the entire table, across all partitions.	A local secondary index lets you query over a single partition, as specified by the partition key value in the query.

Characteristic	Global secondary index	Local secondary index
Read Consistency	Queries on global secondary indexes support eventual consistency only.	When you query a local secondary index, you can choose either eventual consistency or strong consistency.
Provisioned Throughput Consumption	Every global secondary index has its own provisioned throughput settings for read and write activity. Queries or scans on a global secondary index consume capacity units from the index, not from the base table. The same holds true for global secondary index updates due to table writes. A global secondary index associated with global tables consumes write capacity units.	Queries or scans on a local secondary index consume read capacity units from the base table. When you write to a table its local secondary indexes are also updated, and these updates consume write capacity units from the base table. A local secondary index associated with global tables consumes replicated write capacity units.
Projected Attributes	With global secondary index queries or scans, you can only request the attributes that are projected into the index. DynamoDB does not fetch any attributes from the table.	If you query or scan a local secondary index, you can request attributes that are not projected in to the index. DynamoDB automatically fetches those attributes from the table.

If you want to create more than one table with secondary indexes, you must do so sequentially. For example, you would create the first table and wait for it to become `ACTIVE`, create the next table and wait for it to become `ACTIVE`, and so on. If you try to concurrently create more than one table with a secondary index, DynamoDB returns a `LimitExceededException`.

Each secondary index uses the same table class and capacity mode as the base table it is associated with. For each secondary index, you must specify the following:

- The type of index to be created – either a global secondary index or a local secondary index.
- A name for the index. The naming rules for indexes are the same as those for tables, as listed in Quotas in Amazon DynamoDB. The name must be unique for the base table it is associated with, but you can use the same name for indexes that are associated with different base tables.
- The key schema for the index. Every attribute in the index key schema must be a top-level attribute of type `String`, `Number`, or `Binary`. Other data types, including documents and sets, are not allowed. Other requirements for the key schema depend on the type of index:
  - For a global secondary index, the partition key can be any scalar attribute of the base table. A sort key is optional, and it too can be any scalar attribute of the base table.
  - For a local secondary index, the partition key must be the same as the base table's partition key, and the sort key must be a non-key base table attribute.
- Additional attributes, if any, to project from the base table into the index. These attributes are in addition to the table's key attributes, which are automatically projected into every index. You can project attributes of any data type, including scalars, documents, and sets.
- The provisioned throughput settings for the index, if necessary:
  - For a global secondary index, you must specify read and write capacity unit settings. These provisioned throughput settings are independent of the base table's settings.
  - For a local secondary index, you do not need to specify read and write capacity unit settings. Any read and write operations on a local secondary index draw from the provisioned throughput settings of its base table.

For maximum query flexibility, you can create up to 20 global secondary indexes (default quota) and up to 5 local secondary indexes per table.

To get a detailed listing of secondary indexes on a table, use the `DescribeTable` operation. `DescribeTable` returns the name, storage size, and item counts for every secondary index on the table. These values are not updated in real time, but they are refreshed approximately every six hours.

You can access the data in a secondary index using either the `Query` or `Scan` operation. You must specify the name of the base table and the name of the index that you want to use, the attributes to be returned in the results, and any condition expressions or filters that you want to apply. DynamoDB can return the results in ascending or descending order.

When you delete a table, all of the indexes associated with that table are also deleted.

For best practices, see Best practices for using secondary indexes in DynamoDB.

Using Global Secondary Indexes in DynamoDB

Some applications might need to perform many kinds of queries, using a variety of different attributes as query criteria. To support these requirements, you can create one or more *global secondary indexes* and issue `Query` requests against these indexes in Amazon DynamoDB.

Topics

- Scenario: Using a Global Secondary Index
- Attribute projections
- Multi-attribute key schema
- Reading data from a Global Secondary Index
- Data synchronization between tables and Global Secondary Indexes
- Table classes with Global Secondary Index
- Provisioned throughput considerations for Global Secondary Indexes
- Storage considerations for Global Secondary Indexes
- Design patterns
- Managing Global Secondary Indexes in DynamoDB
- Detecting and correcting index key violations in DynamoDB
- Working with Global Secondary Indexes: Java
- Working with Global Secondary Indexes: .NET
- Working with Global Secondary Indexes in DynamoDB using AWS CLI

Scenario: Using a Global Secondary Index

To illustrate, consider a table named `GameScores` that tracks users and scores for a mobile gaming application. Each item in `GameScores` is identified by a partition key (`UserId`) and a sort key (`GameTitle`). The following diagram shows how the items in the table would be organized. (Not all of the attributes are shown.)

GameScores

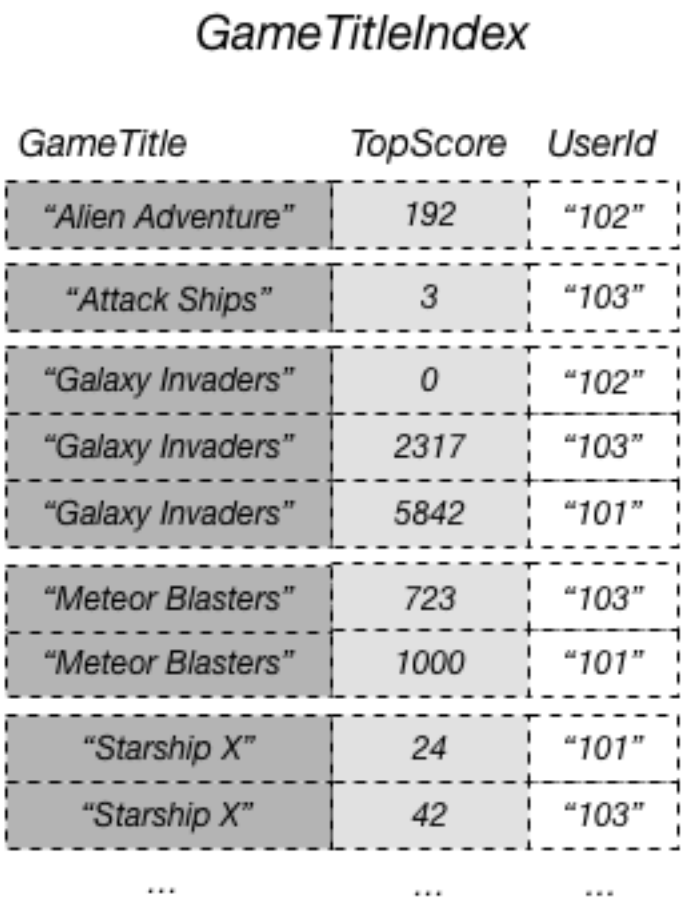
UserId	GameTitle	TopScore	TopScoreDateTime	Wins	Losses	
"101"	"Galaxy Invaders"	5842	"2015-09-15:17:24:31"	21	72	...
"101"	"Meteor Blasters"	1000	"2015-10-22:23:18:01"	12	3	...
"101"	"Starship X"	24	"2015-08-31:13:14:21"	4	9	...
"102"	"Alien Adventure"	192	"2015-07-12:11:07:56"	32	192	...
"102"	"Galaxy Invaders"	0	"2015-09-18:07:33:42"	0	5	...
"103"	"Attack Ships"	3	"2015-10-19:01:13:24"	1	8	...
"103"	"Galaxy Invaders"	2317	"2015-09-11:06:53:00"	40	3	...
"103"	"Meteor Blasters"	723	"2015-10-19:01:13:24"	22	12	...
"103"	"Starship X"	42	"2015-07-11:06:53:00"	4	19	...
...	...	...	...	...	...	

Now suppose that you wanted to write a leaderboard application to display top scores for each game. A query that specified the key attributes (`UserId` and `GameTitle`) would be very efficient. However, if the application needed to retrieve data from `GameScores` based on `GameTitle` only, it would need to use a `Scan` operation. As more items are added to the table, scans of all the data would become slow and inefficient. This makes it difficult to answer questions such as the following:

- What is the top score ever recorded for the game Meteor Blasters?
- Which user had the highest score for Galaxy Invaders?
- What was the highest ratio of wins vs. losses?

To speed up queries on non-key attributes, you can create a global secondary index. A global secondary index contains a selection of attributes from the base table, but they are organized by a primary key that is different from that of the table. The index key does not need to have any of the key attributes from the table. It doesn't even need to have the same key schema as a table.

For example, you could create a global secondary index named `GameTitleIndex`, with a partition key of `GameTitle` and a sort key of `TopScore`. The base table's primary key attributes are always projected into an index, so the `UserId` attribute is also present. The following diagram shows what `GameTitleIndex` index would look like.



Now you can query `GameTitleIndex` and easily obtain the scores for Meteor Blasters. The results are ordered by the sort key values, `TopScore`. If you set the `ScanIndexForward` parameter to false, the results are returned in descending order, so the highest score is returned first.

Every global secondary index must have a partition key, and can have an optional sort key. The index key schema can be different from the base table schema. You could have a table with a simple primary key (partition key), and create a global secondary index with a composite primary key (partition key and sort key)—or vice versa. The index key attributes can consist of any top-level `String`, `Number`, or `Binary` attributes from the base table. Other scalar types, document types, and set types are not allowed.

You can project other base table attributes into the index if you want. When you query the index, DynamoDB can retrieve these projected attributes efficiently. However, global secondary index

queries cannot fetch attributes from the base table. For example, if you query `GameTitleIndex` as shown in the previous diagram, the query could not access any non-key attributes other than `TopScore` (although the key attributes `GameTitle` and `UserId` would automatically be projected).

In a DynamoDB table, each key value must be unique. However, the key values in a global secondary index do not need to be unique. To illustrate, suppose that a game named Comet Quest is especially difficult, with many new users trying but failing to get a score above zero. The following is some data that could represent this.

UserId	GameTitle	TopScore
123	Comet Quest	0
201	Comet Quest	0
301	Comet Quest	0

When this data is added to the `GameScores` table, DynamoDB propagates it to `GameTitleIndex`. If we then query the index using Comet Quest for `GameTitle` and 0 for `TopScore`, the following data is returned.

GameTitle	TopScore	UserId
"Comet Quest"	0	"123"
"Comet Quest"	0	"201"
"Comet Quest"	0	"301"

Only the items with the specified key values appear in the response. Within that set of data, the items are in no particular order.

A global secondary index only tracks data items where its key attributes actually exist. For example, suppose that you added another new item to the `GameScores` table, but only provided the required primary key attributes.

UserId	GameTitle
400	Comet Quest

Because you didn't specify the `TopScore` attribute, DynamoDB would not propagate this item to `GameTitleIndex`. Thus, if you queried `GameScores` for all the Comet Quest items, you would get the following four items.

UserId	GameTitle	TopScore
"123"	"Comet Quest"	0
"201"	"Comet Quest"	0
"301"	"Comet Quest"	0
"400"	"Comet Quest"	

A similar query on `GameTitleIndex` would still return three items, rather than four. This is because the item with the nonexistent `TopScore` is not propagated to the index.

GameTitle	*TopScore*	*UserId*
"Comet Quest"	*0*	*"123"*
"Comet Quest"	*0*	*"201"*
"Comet Quest"	*0*	*"301"*

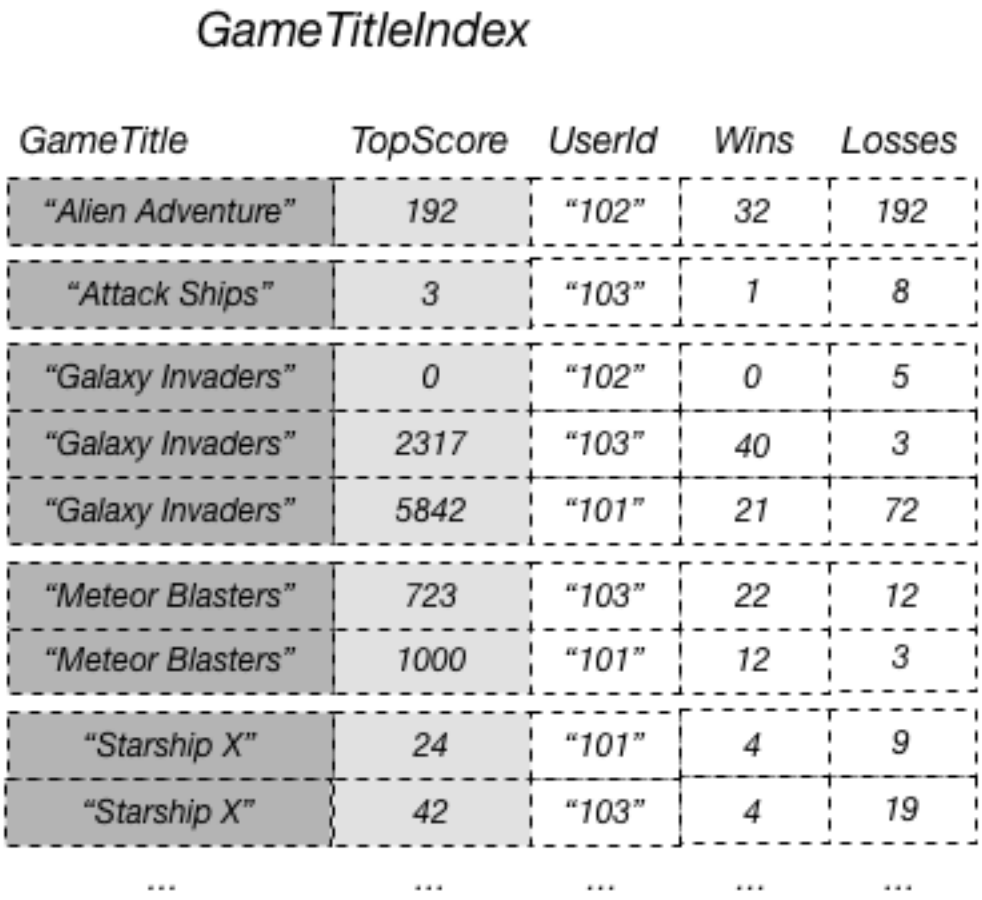
Attribute projections

A *projection* is the set of attributes that is copied from a table into a secondary index. The partition key and sort key of the table are always projected into the index; you can project other attributes to support your application's query requirements. When you query an index, Amazon DynamoDB can access any attribute in the projection as if those attributes were in a table of their own.

When you create a secondary index, you need to specify the attributes that will be projected into the index. DynamoDB provides three different options for this:

- *KEYS_ONLY* – Each item in the index consists only of the table partition key and sort key values, plus the index key values. The `KEYS_ONLY` option results in the smallest possible secondary index.
- *INCLUDE* – In addition to the attributes described in `KEYS_ONLY`, the secondary index will include other non-key attributes that you specify.
- *ALL* – The secondary index includes all of the attributes from the source table. Because all of the table data is duplicated in the index, an `ALL` projection results in the largest possible secondary index.

In the previous diagram, `GameTitleIndex` has only one projected attribute: `UserId`. So while an application can efficiently determine the `UserId` of the top scorers for each game using `GameTitle` and `TopScore` in queries, it can't efficiently determine the highest ratio of wins vs. losses for the top scorers. To do so, the application would have to perform an additional query on the base table to fetch the wins and losses for each of the top scorers. A more efficient way to support queries on this data would be to project these attributes from the base table into the global secondary index, as shown in this diagram.



Because the non-key attributes `Wins` and `Losses` are projected into the index, an application can determine the wins vs. losses ratio for any game, or for any combination of game and user ID.

When you choose the attributes to project into a global secondary index, you must consider the tradeoff between provisioned throughput costs and storage costs:

- If you need to access just a few attributes with the lowest possible latency, consider projecting only those attributes into a global secondary index. The smaller the index, the less that it costs to store it, and the less your write costs are.
- If your application frequently accesses some non-key attributes, you should consider projecting those attributes into a global secondary index. The additional storage costs for the global secondary index offset the cost of performing frequent table scans.
- If you need to access most of the non-key attributes on a frequent basis, you can project these attributes—or even the entire base table— into a global secondary index. This gives you maximum flexibility. However, your storage cost would increase, or even double.
- If your application needs to query a table infrequently, but must perform many writes or updates against the data in the table, consider projecting `KEYS_ONLY`. The global secondary index would be of minimal size, but would still be available when needed for query activity.

Multi-attribute key schema

Global Secondary Indexes support multi-attribute keys, allowing you to compose partition keys and sort keys from multiple attributes. With multi-attribute keys, you can create a partition key from up to four attributes and a sort key from up to four attributes, for a total of up to eight attributes per key schema.

Multi-attribute keys simplify your data model by eliminating the need to manually concatenate attributes into synthetic keys. Instead of creating composite strings like `TOURNAMENT#WINTER2024#REGION#NA-EAST`, you can use the natural attributes from your domain model directly. DynamoDB handles the composite key logic automatically, hashing multiple partition key attributes together for data distribution and maintaining hierarchical sort order across multiple sort key attributes.

For example, consider a gaming tournament system where you want to organize matches by tournament and region. With multi-attribute keys, you can define your partition key as two separate attributes: `tournamentId` and `region`. Similarly, you can define your sort key using multiple attributes like `round`, `bracket`, and `matchId` to create a natural hierarchy. This approach keeps your data typed and your code clean, without string manipulation or parsing.

When you query a global secondary index with multi-attribute keys, you must specify all partition key attributes using equality conditions. For sort key attributes, you can query them left-to-right in the order they're defined in the key schema. This means you can query the first sort key attribute alone, the first two attributes together, or all attributes together, but you cannot skip attributes in the middle. Inequality conditions such as >, <, `BETWEEN`, or `begins_with()` must be the last condition in your query.

Multi-attribute keys work particularly well when creating global secondary indexes on existing tables. You can use attributes that already exist in your table without backfilling synthetic keys across your data. This makes it straightforward to add new query patterns to your application by creating indexes that reorganize your data using different attribute combinations.

Each attribute in a multi-attribute key can have its own data type: `String` (S), `Number` (N), or `Binary` (B). When choosing data types, consider that `Number` attributes sort numerically without requiring zero-padding, while `String` attributes sort lexicographically. For example, if you use a `Number` type for a score attribute, the values 5, 50, 500, and 1000 sort in natural numeric order. The same values as `String` type would sort as "1000", "5", "50", "500" unless you pad them with leading zeros.

When designing multi-attribute keys, order your attributes from most general to most specific. For partition keys, combine attributes that are always queried together and that provide good data distribution. For sort keys, place frequently queried attributes first in the hierarchy to maximize query flexibility. This ordering allows you to query at any level of granularity that matches your access patterns.

See the Multi-attribute keys for implementation examples.

Reading data from a Global Secondary Index

You can retrieve items from a global secondary index using the `Query` and `Scan` operations. The `GetItem` and `BatchGetItem` operations can't be used on a global secondary index.

Querying a Global Secondary Index

You can use the `Query` operation to access one or more items in a global secondary index. The query must specify the name of the base table and the name of the index that you want to use, the attributes to be returned in the query results, and any query conditions that you want to apply. DynamoDB can return the results in ascending or descending order.

Consider the following data returned from a `Query` that requests gaming data for a leaderboard application.

```
{
    "TableName": "GameScores",
    "IndexName": "GameTitleIndex",
    "KeyConditionExpression": "GameTitle = :v_title",
    "ExpressionAttributeValues": {
        ":v_title": {"S": "Meteor Blasters"}
    },
    "ProjectionExpression": "UserId, TopScore",
    "ScanIndexForward": false
}
```

In this query:

- DynamoDB accesses *GameTitleIndex*, using the *GameTitle* partition key to locate the index items for Meteor Blasters. All of the index items with this key are stored adjacent to each other for rapid retrieval.
- Within this game, DynamoDB uses the index to access all of the user IDs and top scores for this game.
- The results are returned, sorted in descending order because the `ScanIndexForward` parameter is set to false.

Scanning a Global Secondary Index

You can use the `Scan` operation to retrieve all of the data from a global secondary index. You must provide the base table name and the index name in the request. With a `Scan`, DynamoDB reads all of the data in the index and returns it to the application. You can also request that only some of the data be returned, and that the remaining data should be discarded. To do this, use the `FilterExpression` parameter of the `Scan` operation. For more information, see Filter expressions for scan.

Data synchronization between tables and Global Secondary Indexes

DynamoDB automatically synchronizes each global secondary index with its base table. When an application writes or deletes items in a table, any global secondary indexes on that table are updated asynchronously, using an eventually consistent model. Applications never write directly

to an index. However, it is important that you understand the implications of how DynamoDB maintains these indexes.

Global secondary indexes inherit the read/write capacity mode from the base table. For more information, see Considerations when switching capacity modes in DynamoDB.

When you create a global secondary index, you specify one or more index key attributes and their data types. This means that whenever you write an item to the base table, the data types for those attributes must match the index key schema's data types. In the case of `GameTitleIndex`, the `GameTitle` partition key in the index is defined as a `String` data type. The `TopScore` sort key in the index is of type `Number`. If you try to add an item to the `GameScores` table and specify a different data type for either `GameTitle` or `TopScore`, DynamoDB returns a `ValidationException` because of the data type mismatch.

When you put or delete items in a table, the global secondary indexes on that table are updated in an eventually consistent fashion. Changes to the table data are propagated to the global secondary indexes within a fraction of a second, under normal conditions. However, in some unlikely failure scenarios, longer propagation delays might occur. Because of this, your applications need to anticipate and handle situations where a query on a global secondary index returns results that are not up to date.

If you write an item to a table, you don't have to specify the attributes for any global secondary index sort key. Using `GameTitleIndex` as an example, you would not need to specify a value for the `TopScore` attribute to write a new item to the `GameScores` table. In this case, DynamoDB does not write any data to the index for this particular item.

A table with many global secondary indexes incurs higher costs for write activity than tables with fewer indexes. For more information, see Provisioned throughput considerations for Global Secondary Indexes.

Table classes with Global Secondary Index

A global secondary index will always use the same table class as its base table. Any time a new global secondary index is added for a table, the new index will use the same table class as its base table. When a table's table class is updated, all associated global secondary indexes are updated as well.

Provisioned throughput considerations for Global Secondary Indexes

When you create a global secondary index on a provisioned mode table, you must specify read and write capacity units for the expected workload on that index. The provisioned throughput settings of a global secondary index are separate from those of its base table. A `Query` operation on a global secondary index consumes read capacity units from the index, not the base table. When you put, update or delete items in a table, the global secondary indexes on that table are also updated. These index updates consume write capacity units from the index, not from the base table.

For example, if you `Query` a global secondary index and exceed its provisioned read capacity, your request will be throttled. If you perform heavy write activity on the table, but a global secondary index on that table has insufficient write capacity, the write activity on the table will be throttled.

> **Important**
>
> To avoid potential throttling, the provisioned write capacity for a global secondary index should be equal or greater than the write capacity of the base table because new updates write to both the base table and global secondary index.

To view the provisioned throughput settings for a global secondary index, use the `DescribeTable` operation. Detailed information about all of the table's global secondary indexes is returned.

Read capacity units

Global secondary indexes support eventually consistent reads, each of which consume one half of a read capacity unit. This means that a single global secondary index query can retrieve up to 2 × 4 KB = 8 KB per read capacity unit.

For global secondary index queries, DynamoDB calculates the provisioned read activity in the same way as it does for queries against tables. The only difference is that the calculation is based on the sizes of the index entries, rather than the size of the item in the base table. The number of read capacity units is the sum of all projected attribute sizes across all of the items returned. The result is then rounded up to the next 4 KB boundary. For more information about how DynamoDB calculates provisioned throughput usage, see DynamoDB provisioned capacity mode.

The maximum size of the results returned by a `Query` operation is 1 MB. This includes the sizes of all the attribute names and values across all of the items returned.

For example, consider a global secondary index where each item contains 2,000 bytes of data. Now suppose that you `Query` this index and that the query's `KeyConditionExpression` matches eight items. The total size of the matching items is 2,000 bytes × 8 items = 16,000 bytes. This result is then rounded up to the nearest 4 KB boundary. Because global secondary index queries are eventually consistent, the total cost is 0.5 × (16 KB / 4 KB), or 2 read capacity units.

Write capacity units

When an item in a table is added, updated, or deleted, and a global secondary index is affected by this, the global secondary index consumes provisioned write capacity units for the operation. The total provisioned throughput cost for a write consists of the sum of write capacity units consumed by writing to the base table and those consumed by updating the global secondary indexes. If a write to a table does not require a global secondary index update, no write capacity is consumed from the index.

For a table write to succeed, the provisioned throughput settings for the table and all of its global secondary indexes must have enough write capacity to accommodate the write. Otherwise, the write to the table is throttled.

> ⚠ **Important**
>
> When creating a Global Secondary Index (GSI), write operations to the base table can be throttled if the GSI activity resulting from writes to the base table exceeds the GSI's provisioned write capacity. This throttling affects all write operations, from indexing process to potentially disrupting your production workloads. For more information, see Troubleshooting throttling in Amazon DynamoDB.

The cost of writing an item to a global secondary index depends on several factors:

- If you write a new item to the table that defines an indexed attribute, or you update an existing item to define a previously undefined indexed attribute, one write operation is required to put the item into the index.
- If an update to the table changes the value of an indexed key attribute (from A to B), two writes are required, one to delete the previous item from the index and another write to put the new item into the index.
- If an item was present in the index, but a write to the table caused the indexed attribute to be deleted, one write is required to delete the old item projection from the index.

- If an item is not present in the index before or after the item is updated, there is no additional write cost for the index.
- If an update to the table only changes the value of projected attributes in the index key schema, but does not change the value of any indexed key attribute, one write is required to update the values of the projected attributes into the index.

All of these factors assume that the size of each item in the index is less than or equal to the 1 KB item size for calculating write capacity units. Larger index entries require additional write capacity units. You can minimize your write costs by considering which attributes your queries will need to return and projecting only those attributes into the index.

Storage considerations for Global Secondary Indexes

When an application writes an item to a table, DynamoDB automatically copies the correct subset of attributes to any global secondary indexes in which those attributes should appear. Your AWS account is charged for storage of the item in the base table and also for storage of attributes in any global secondary indexes on that table.

The amount of space used by an index item is the sum of the following:

- The size in bytes of the base table primary key (partition key and sort key)
- The size in bytes of the index key attribute
- The size in bytes of the projected attributes (if any)
- 100 bytes of overhead per index item

To estimate the storage requirements for a global secondary index, you can estimate the average size of an item in the index and then multiply by the number of items in the base table that have the global secondary index key attributes.

If a table contains an item where a particular attribute(s) is not defined, but that attribute is defined as an index partition key or sort key, DynamoDB doesn't write any data for that item to the index.

Design patterns

Design patterns provide proven solutions to common challenges when working with global secondary indexes. These patterns help you build efficient, scalable applications by showing you how to structure your indexes for specific use cases.

Each pattern includes a complete implementation guide with code examples, best practices, and real-world use cases to help you apply the pattern to your own applications.

Topics

- Multi-attribute keys pattern

Multi-attribute keys pattern

Overview

Multi-attribute keys allow you to create Global Secondary Index (GSI) partition and sort keys composed of up to four attributes each. This reduces client-side code and makes it easier to initially model data and add new access patterns later.

Consider a common scenario: to create a GSI that queries items by multiple hierarchical attributes, you would traditionally need to create synthetic keys by concatenating values. For example, in a gaming app, to query tournament matches by tournament, region, and round, you might create a synthetic GSI partition key like TOURNAMENT#WINTER2024#REGION#NA-EAST and a synthetic sort key like ROUND#SEMIFINALS#BRACKET#UPPER. This approach works, but requires string concatenation when writing data, parsing when reading, and backfilling synthetic keys across all existing items if you're adding the GSI to an existing table. This makes code more cluttered and challenging to maintain type safety on individual key components.

Multi-attribute keys solve this problem for GSIs. You define your GSI partition key using multiple existing attributes like tournamentId and region. DynamoDB handles the composite key logic automatically, hashing them together for data distribution. You write items using natural attributes from your domain model, and the GSI automatically indexes them. No concatenation, no parsing, no backfilling. Your code stays clean, your data stays typed, and your queries stay simple. This approach is particularly useful when you have hierarchical data with natural attribute groupings (like tournament → region → round, or organization → department → team).

Application example

This guide walks through building a tournament match tracking system for an esports platform. The platform needs to efficiently query matches across multiple dimensions: by tournament and region for bracket management, by player for match history, and by date for scheduling.

Data model

In this walkthrough, the tournament match tracking system supports three primary access patterns, each requiring a different key structure:

Access pattern 1: Look up a specific match by its unique ID

- **Solution:** Base table with `matchId` as partition key

Access pattern 2: Query all matches for a specific tournament and region, optionally filtering by round, bracket, or match

- **Solution:** Global Secondary Index with multi-attribute partition key (`tournamentId` + `region`) and multi-attribute sort key (`round` + `bracket` + `matchId`)
- **Example queries:** "All WINTER2024 matches in NA-EAST region" or "All SEMIFINALS matches in UPPER bracket for WINTER2024/NA-EAST"

Access pattern 3: Query a player's match history, optionally filtering by date range or tournament round

- **Solution:** Global Secondary Index with single partition key (`player1Id`) and multi-attribute sort key (`matchDate` + `round`)
- **Example queries:** "All matches for player 101" or "Player 101's matches in January 2024"

The key difference between traditional and multi-attribute approaches becomes clear when examining the item structure:

Traditional Global Secondary Index approach (concatenated keys):

```
// Manual concatenation required for GSI keys
const item = {
    matchId: 'match-001',                                          // Base table PK
    tournamentId: 'WINTER2024',
    region: 'NA-EAST',
    round: 'SEMIFINALS',
    bracket: 'UPPER',
    player1Id: '101',
    // Synthetic keys needed for GSI
    GSI_PK: `TOURNAMENT#${tournamentId}#REGION#${region}`,       // Must concatenate
```

```
    GSI_SK: `${round}#${bracket}#${matchId}`,                    // Must concatenate
    // ... other attributes
};
```

Multi-attribute Global Secondary Index approach (native keys):

```
// Use existing attributes directly - no concatenation needed
const item = {
    matchId: 'match-001',                                          // Base table PK
    tournamentId: 'WINTER2024',
    region: 'NA-EAST',
    round: 'SEMIFINALS',
    bracket: 'UPPER',
    player1Id: '101',
    matchDate: '2024-01-18',
    // No synthetic keys needed - GSI uses existing attributes directly
    // ... other attributes
};
```

With multi-attribute keys, you write items once with natural domain attributes. DynamoDB automatically indexes them across multiple GSIs without requiring synthetic concatenated keys.

Base table schema:

- Partition key: `matchId` (1 attribute)

Global Secondary Index Schema (TournamentRegionIndex with multi-attribute keys):

- Partition key: `tournamentId`, `region` (2 attributes)
- Sort key: `round`, `bracket`, `matchId` (3 attributes)

Global Secondary Index Schema (PlayerMatchHistoryIndex with multi-attribute keys):

- Partition key: `player1Id` (1 attribute)
- Sort key: `matchDate`, `round` (2 attributes)

Base table: TournamentMatches

matchId (PK)	tournamtId	region	round	bracket	player1I	player2I	matchDa	winner	score
match-0	WINTER2 4	NA-EAST	FINALS	CHAMPI HIP	101	103	2024-01 0	101	3-1
match-0	WINTER2 4	NA-EAST	SEMIFIN S	UPPER	101	105	2024-01 8	101	3-2
match-0	WINTER2 4	NA-EAST	SEMIFIN S	UPPER	103	107	2024-01 8	103	3-0
match-0	WINTER2 4	NA-EAST	QUARTE NALS	UPPER	101	109	2024-01 5	101	3-1
match-0	WINTER2 4	NA-WEST	FINALS	CHAMPI HIP	102	104	2024-01 0	102	3-2
match-0	WINTER2 4	NA-WEST	SEMIFIN S	UPPER	102	106	2024-01 8	102	3-1
match-0	SPRING2 4	NA-EAST	QUARTE NALS	UPPER	101	108	2024-03 5	101	3-0
match-0	SPRING2 4	NA-EAST	QUARTE NALS	LOWER	103	110	2024-03 5	103	3-2

GSI: TournamentRegionIndex (multi-attribute keys)

tournamtId (PK)	region (PK)	round (SK)	bracket (SK)	matchId (SK)	player1I	player2I	matchDa	winner	score
WINTER2 4	NA-EAST	FINALS	CHAMPI HIP	match-0	101	103	2024-01 0	101	3-1

tournamtId (PK)	region (PK)	round (SK)	bracket (SK)	matchId (SK)	player1I	player2I	matchDa	winner	score
WINTER2 4	NA-EAST	QUARTE NALS	UPPER	match-0	101	109	2024-01 5	101	3-1
WINTER2 4	NA-EAST	SEMIFIN S	UPPER	match-0	101	105	2024-01 8	101	3-2
WINTER2 4	NA-EAST	SEMIFIN S	UPPER	match-0	103	107	2024-01 8	103	3-0
WINTER2 4	NA-WEST	FINALS	CHAMPI HIP	match-0	102	104	2024-01 0	102	3-2
WINTER2 4	NA-WEST	SEMIFIN S	UPPER	match-0	102	106	2024-01 8	102	3-1
SPRING2 4	NA-EAST	QUARTE NALS	LOWER	match-0	103	110	2024-03 5	103	3-2
SPRING2 4	NA-EAST	QUARTE NALS	UPPER	match-0	101	108	2024-03 5	101	3-0

GSI: PlayerMatchHistoryIndex (multi-attribute keys)

player1I (PK)	matchDa (SK)	round (SK)	tournamtId	region	bracket	matchId	player2I	winner	score
101	2024-01 5	QUARTE NALS	WINTER2 4	NA-EAST	UPPER	match-0	109	101	3-1
101	2024-01 8	SEMIFIN S	WINTER2 4	NA-EAST	UPPER	match-0	105	101	3-2
101	2024-01 0	FINALS	WINTER2 4	NA-EAST	CHAMPI HIP	match-0	103	101	3-1

player1I (PK)	matchDa (SK)	round (SK)	tournam tId	region	bracket	matchId	player2I	winner	score
101	2024-03 5	QUARTE NALS	SPRING2 4	NA-EAST	UPPER	match-0	108	101	3-0
102	2024-01 8	SEMIFIN S	WINTER2 4	NA-WEST	UPPER	match-0	106	102	3-1
102	2024-01 0	FINALS	WINTER2 4	NA-WEST	CHAMPI HIP	match-0	104	102	3-2
103	2024-01 8	SEMIFIN S	WINTER2 4	NA-EAST	UPPER	match-0	107	103	3-0
103	2024-03 5	QUARTE NALS	SPRING2 4	NA-EAST	LOWER	match-0	110	103	3-2

Prerequisites

Before you begin, ensure you have:

Account and permissions

- An active AWS account (create one here if needed)
- IAM permissions for DynamoDB operations:
  - `dynamodb:CreateTable`
  - `dynamodb:DeleteTable`
  - `dynamodb:DescribeTable`
  - `dynamodb:PutItem`
  - `dynamodb:Query`
  - `dynamodb:BatchWriteItem`

> **Note**
>
> **Security Note:** For production use, create a custom IAM policy with only the permissions you need. For this tutorial, you can use the AWS managed policy `AmazonDynamoDBFullAccessV2`.

Development Environment

- Node.js installed on your machine
- AWS credentials configured using one of these methods:

Option 1: AWS CLI

```
aws configure
```

Option 2: Environment Variables

```
export AWS_ACCESS_KEY_ID=your_access_key_here
export AWS_SECRET_ACCESS_KEY=your_secret_key_here
export AWS_DEFAULT_REGION=us-east-1
```

Install Required Packages

```
npm install @aws-sdk/client-dynamodb @aws-sdk/lib-dynamodb
```

Implementation

Step 1: Create table with GSIs using multi-attribute keys

Create a table with a simple base key structure and GSIs that use multi-attribute keys.

Code example

```
import { DynamoDBClient, CreateTableCommand } from "@aws-sdk/client-dynamodb";

const client = new DynamoDBClient({ region: 'us-west-2' });

const response = await client.send(new CreateTableCommand({
    TableName: 'TournamentMatches',
```

```
    // Base table: Simple partition key
    KeySchema: [
        { AttributeName: 'matchId', KeyType: 'HASH' }              // Simple PK
    ],

    AttributeDefinitions: [
        { AttributeName: 'matchId', AttributeType: 'S' },
        { AttributeName: 'tournamentId', AttributeType: 'S' },
        { AttributeName: 'region', AttributeType: 'S' },
        { AttributeName: 'round', AttributeType: 'S' },
        { AttributeName: 'bracket', AttributeType: 'S' },
        { AttributeName: 'player1Id', AttributeType: 'S' },
        { AttributeName: 'matchDate', AttributeType: 'S' }
    ],

    // GSIs with multi-attribute keys
    GlobalSecondaryIndexes: [
        {
            IndexName: 'TournamentRegionIndex',
            KeySchema: [
                { AttributeName: 'tournamentId', KeyType: 'HASH' },    // GSI PK
attribute 1
                { AttributeName: 'region', KeyType: 'HASH' },          // GSI PK
attribute 2
                { AttributeName: 'round', KeyType: 'RANGE' },          // GSI SK
attribute 1
                { AttributeName: 'bracket', KeyType: 'RANGE' },        // GSI SK
attribute 2
                { AttributeName: 'matchId', KeyType: 'RANGE' }         // GSI SK
attribute 3
            ],
            Projection: { ProjectionType: 'ALL' }
        },
        {
            IndexName: 'PlayerMatchHistoryIndex',
            KeySchema: [
                { AttributeName: 'player1Id', KeyType: 'HASH' },       // GSI PK
                { AttributeName: 'matchDate', KeyType: 'RANGE' },      // GSI SK
attribute 1
                { AttributeName: 'round', KeyType: 'RANGE' }           // GSI SK
attribute 2
            ],
            Projection: { ProjectionType: 'ALL' }
```

```
        }
    ],

    BillingMode: 'PAY_PER_REQUEST'
}));

console.log("Table with multi-attribute GSI keys created successfully");
```

Key design decisions:

Base table: The base table uses a simple `matchId` partition key for direct match lookups, keeping the base table structure straightforward while the GSIs provide the complex query patterns.

TournamentRegionIndex Global Secondary Index: The `TournamentRegionIndex` Global Secondary Index uses `tournamentId` + `region` as a multi-attribute partition key, creating tournament-region isolation where data is distributed by the hash of both attributes combined, enabling efficient queries within a specific tournament-region context. The multi-attribute sort key (`round` + `bracket` + `matchId`) provides hierarchical sorting that supports queries at any level of the hierarchy with natural ordering from general (round) to specific (match ID).

PlayerMatchHistoryIndex Global Secondary Index: The `PlayerMatchHistoryIndex` Global Secondary Index reorganizes data by player using `player1Id` as the partition key, enabling cross-tournament queries for a specific player. The multi-attribute sort key (`matchDate` + `round`) provides chronological ordering with the ability to filter by date ranges or specific tournament rounds.

Step 2: Insert data with native attributes

Add tournament match data using natural attributes. The GSI will automatically index these attributes without requiring synthetic keys.

Code example

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
import { DynamoDBDocumentClient, PutCommand } from "@aws-sdk/lib-dynamodb";

const client = new DynamoDBClient({ region: 'us-west-2' });
const docClient = DynamoDBDocumentClient.from(client);

// Tournament match data - no synthetic keys needed for GSIs
const matches = [
    // Winter 2024 Tournament, NA-EAST region
```

```
    {
        matchId: 'match-001',
        tournamentId: 'WINTER2024',
        region: 'NA-EAST',
        round: 'FINALS',
        bracket: 'CHAMPIONSHIP',
        player1Id: '101',
        player2Id: '103',
        matchDate: '2024-01-20',
        winner: '101',
        score: '3-1'
    },
    {
        matchId: 'match-002',
        tournamentId: 'WINTER2024',
        region: 'NA-EAST',
        round: 'SEMIFINALS',
        bracket: 'UPPER',
        player1Id: '101',
        player2Id: '105',
        matchDate: '2024-01-18',
        winner: '101',
        score: '3-2'
    },
    {
        matchId: 'match-003',
        tournamentId: 'WINTER2024',
        region: 'NA-EAST',
        round: 'SEMIFINALS',
        bracket: 'UPPER',
        player1Id: '103',
        player2Id: '107',
        matchDate: '2024-01-18',
        winner: '103',
        score: '3-0'
    },
    {
        matchId: 'match-004',
        tournamentId: 'WINTER2024',
        region: 'NA-EAST',
        round: 'QUARTERFINALS',
        bracket: 'UPPER',
        player1Id: '101',
        player2Id: '109',
```

```
        matchDate: '2024-01-15',
        winner: '101',
        score: '3-1'
    },

    // Winter 2024 Tournament, NA-WEST region
    {
        matchId: 'match-005',
        tournamentId: 'WINTER2024',
        region: 'NA-WEST',
        round: 'FINALS',
        bracket: 'CHAMPIONSHIP',
        player1Id: '102',
        player2Id: '104',
        matchDate: '2024-01-20',
        winner: '102',
        score: '3-2'
    },
    {
        matchId: 'match-006',
        tournamentId: 'WINTER2024',
        region: 'NA-WEST',
        round: 'SEMIFINALS',
        bracket: 'UPPER',
        player1Id: '102',
        player2Id: '106',
        matchDate: '2024-01-18',
        winner: '102',
        score: '3-1'
    },

    // Spring 2024 Tournament, NA-EAST region
    {
        matchId: 'match-007',
        tournamentId: 'SPRING2024',
        region: 'NA-EAST',
        round: 'QUARTERFINALS',
        bracket: 'UPPER',
        player1Id: '101',
        player2Id: '108',
        matchDate: '2024-03-15',
        winner: '101',
        score: '3-0'
    },
```

```
    {
        matchId: 'match-008',
        tournamentId: 'SPRING2024',
        region: 'NA-EAST',
        round: 'QUARTERFINALS',
        bracket: 'LOWER',
        player1Id: '103',
        player2Id: '110',
        matchDate: '2024-03-15',
        winner: '103',
        score: '3-2'
    }
];

// Insert all matches
for (const match of matches) {
    await docClient.send(new PutCommand({
        TableName: 'TournamentMatches',
        Item: match
    }));

    console.log(`Added: ${match.matchId} - ${match.tournamentId}/${match.region} -
 ${match.round} ${match.bracket}`);
}

console.log(`\nInserted ${matches.length} tournament matches`);
console.log("No synthetic keys created - GSIs use native attributes automatically");
```

Data structure explained:

Natural attribute usage: Each attribute represents a real tournament concept with no string concatenation or parsing required, providing direct mapping to the domain model.

Automatic Global Secondary Index indexing: The GSIs automatically index items using the existing attributes (`tournamentId`, `region`, `round`, `bracket`, `matchId` for TournamentRegionIndex and `player1Id`, `matchDate`, `round` for PlayerMatchHistoryIndex) without requiring synthetic concatenated keys.

No backfilling needed: When you add a new Global Secondary Index with multi-attribute keys to an existing table, DynamoDB automatically indexes all existing items using their natural attributes —no need to update items with synthetic keys.

Step 3: Query TournamentRegionIndex Global Secondary Index with all partition key attributes

This example queries the TournamentRegionIndex Global Secondary Index which has a multi-attribute partition key (`tournamentId` + `region`). All partition key attributes must be specified with equality conditions in queries—you cannot query with just `tournamentId` alone or use inequality operators on partition key attributes.

Code example

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
import { DynamoDBDocumentClient, QueryCommand } from "@aws-sdk/lib-dynamodb";

const client = new DynamoDBClient({ region: 'us-west-2' });
const docClient = DynamoDBDocumentClient.from(client);

// Query GSI: All matches for WINTER2024 tournament in NA-EAST region
const response = await docClient.send(new QueryCommand({
    TableName: 'TournamentMatches',
    IndexName: 'TournamentRegionIndex',
    KeyConditionExpression: 'tournamentId = :tournament AND #region = :region',
    ExpressionAttributeNames: {
        '#region': 'region',  // 'region' is a reserved keyword
        '#tournament': 'tournament'
    },
    ExpressionAttributeValues: {
        ':tournament': 'WINTER2024',
        ':region': 'NA-EAST'
    }
}));

console.log(`Found ${response.Items.length} matches for WINTER2024/NA-EAST:\n`);
response.Items.forEach(match => {
    console.log(`  ${match.round} | ${match.bracket} | ${match.matchId}`);
    console.log(`    Players: ${match.player1Id} vs ${match.player2Id}`);
    console.log(`    Winner: ${match.winner}, Score: ${match.score}\n`);
});
```

Expected output:

```
Found 4 matches for WINTER2024/NA-EAST:

  FINALS | CHAMPIONSHIP | match-001
    Players: 101 vs 103
```

```
    Winner: 101, Score: 3-1

  QUARTERFINALS | UPPER | match-004
    Players: 101 vs 109
    Winner: 101, Score: 3-1

  SEMIFINALS | UPPER | match-002
    Players: 101 vs 105
    Winner: 101, Score: 3-2

  SEMIFINALS | UPPER | match-003
    Players: 103 vs 107
    Winner: 103, Score: 3-0
```

Invalid queries:

```
// Missing region attribute
KeyConditionExpression: 'tournamentId = :tournament'

// Using inequality on partition key attribute
KeyConditionExpression: 'tournamentId = :tournament AND #region > :region'
```

Performance: Multi-attribute partition keys are hashed together, providing the same O(1) lookup performance as single-attribute keys.

Step 4: Query Global Secondary Index sort keys left-to-right

Sort key attributes must be queried left-to-right in the order they're defined in the Global Secondary Index. This example demonstrates querying the TournamentRegionIndex at different hierarchy levels: filtering by just `round`, by `round` + `bracket`, or by all three sort key attributes. You cannot skip attributes in the middle—for example, you cannot query by `round` and `matchId` while skipping `bracket`.

Code example

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
import { DynamoDBDocumentClient, QueryCommand } from "@aws-sdk/lib-dynamodb";

const client = new DynamoDBClient({ region: 'us-west-2' });
const docClient = DynamoDBDocumentClient.from(client);

// Query 1: Filter by first sort key attribute (round)
```

```
console.log("Query 1: All SEMIFINALS matches");
const query1 = await docClient.send(new QueryCommand({
    TableName: 'TournamentMatches',
    IndexName: 'TournamentRegionIndex',
    KeyConditionExpression: 'tournamentId = :tournament AND #region = :region AND round
 = :round',
    ExpressionAttributeNames: {
        '#region': 'region'  // 'region' is a reserved keyword
    },
    ExpressionAttributeValues: {
        ':tournament': 'WINTER2024',
        ':region': 'NA-EAST',
        ':round': 'SEMIFINALS'
    }
}));
console.log(`  Found ${query1.Items.length} matches\n`);

// Query 2: Filter by first two sort key attributes (round + bracket)
console.log("Query 2: SEMIFINALS UPPER bracket matches");
const query2 = await docClient.send(new QueryCommand({
    TableName: 'TournamentMatches',
    IndexName: 'TournamentRegionIndex',
    KeyConditionExpression: 'tournamentId = :tournament AND #region = :region AND round
 = :round AND bracket = :bracket',
    ExpressionAttributeNames: {
        '#region': 'region'  // 'region' is a reserved keyword
    },
    ExpressionAttributeValues: {
        ':tournament': 'WINTER2024',
        ':region': 'NA-EAST',
        ':round': 'SEMIFINALS',
        ':bracket': 'UPPER'
    }
}));
console.log(`  Found ${query2.Items.length} matches\n`);

// Query 3: Filter by all three sort key attributes (round + bracket + matchId)
console.log("Query 3: Specific match in SEMIFINALS UPPER bracket");
const query3 = await docClient.send(new QueryCommand({
    TableName: 'TournamentMatches',
    IndexName: 'TournamentRegionIndex',
    KeyConditionExpression: 'tournamentId = :tournament AND #region = :region AND round
 = :round AND bracket = :bracket AND matchId = :matchId',
    ExpressionAttributeNames: {
```

```
        '#region': 'region'  // 'region' is a reserved keyword
    },
    ExpressionAttributeValues: {
        ':tournament': 'WINTER2024',
        ':region': 'NA-EAST',
        ':round': 'SEMIFINALS',
        ':bracket': 'UPPER',
        ':matchId': 'match-002'
    }
}));
console.log(`  Found ${query3.Items.length} matches\n`);

// Query 4: INVALID - skipping round
console.log("Query 4: Attempting to skip first sort key attribute (WILL FAIL)");
try {
    const query4 = await docClient.send(new QueryCommand({
        TableName: 'TournamentMatches',
        IndexName: 'TournamentRegionIndex',
        KeyConditionExpression: 'tournamentId = :tournament AND #region = :region AND
 bracket = :bracket',
        ExpressionAttributeNames: {
            '#region': 'region'  // 'region' is a reserved keyword
        },
        ExpressionAttributeValues: {
            ':tournament': 'WINTER2024',
            ':region': 'NA-EAST',
            ':bracket': 'UPPER'
        }
    }));
} catch (error) {
    console.log(`  Error: ${error.message}`);
    console.log(`  Cannot skip sort key attributes - must query left-to-right\n`);
}
```

Expected output:

```
Query 1: All SEMIFINALS matches
  Found 2 matches

Query 2: SEMIFINALS UPPER bracket matches
  Found 2 matches

Query 3: Specific match in SEMIFINALS UPPER bracket
```

```
  Found 1 matches

Query 4: Attempting to skip first sort key attribute (WILL FAIL)
  Error: Query key condition not supported
  Cannot skip sort key attributes - must query left-to-right
```

Left-to-right query rules: You must query attributes in order from left to right, without skipping any.

Valid patterns:

- First attribute only: `round = 'SEMIFINALS'`
- First two attributes: `round = 'SEMIFINALS' AND bracket = 'UPPER'`
- All three attributes: `round = 'SEMIFINALS' AND bracket = 'UPPER' AND matchId = 'match-002'`

Invalid patterns:

- Skipping the first attribute: `bracket = 'UPPER'` (skips round)
- Querying out of order: `matchId = 'match-002' AND round = 'SEMIFINALS'`
- Leaving gaps: `round = 'SEMIFINALS' AND matchId = 'match-002'` (skips bracket)

> **Note**
>
> **Design tip:** Order sort key attributes from most general to most specific to maximize query flexibility.

Step 5: Use inequality conditions on Global Secondary Index sort keys

Inequality conditions must be the last condition in your query. This example demonstrates using comparison operators (>=, `BETWEEN`) and prefix matching (`begins_with()`) on sort key attributes. Once you use an inequality operator, you cannot add any additional sort key conditions after it—the inequality must be the final condition in your key condition expression.

Code example

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
```

```
import { DynamoDBDocumentClient, QueryCommand } from "@aws-sdk/lib-dynamodb";

const client = new DynamoDBClient({ region: 'us-west-2' });
const docClient = DynamoDBDocumentClient.from(client);

// Query 1: Round comparison (inequality on first sort key attribute)
console.log("Query 1: Matches from QUARTERFINALS onwards");
const query1 = await docClient.send(new QueryCommand({
    TableName: 'TournamentMatches',
    IndexName: 'TournamentRegionIndex',
    KeyConditionExpression: 'tournamentId = :tournament AND #region = :region AND round
 >= :round',
    ExpressionAttributeNames: {
        '#region': 'region'  // 'region' is a reserved keyword
    },
    ExpressionAttributeValues: {
        ':tournament': 'WINTER2024',
        ':region': 'NA-EAST',
        ':round': 'QUARTERFINALS'
    }
}));
console.log(`  Found ${query1.Items.length} matches\n`);

// Query 2: Round range with BETWEEN
console.log("Query 2: Matches between QUARTERFINALS and SEMIFINALS");
const query2 = await docClient.send(new QueryCommand({
    TableName: 'TournamentMatches',
    IndexName: 'TournamentRegionIndex',
    KeyConditionExpression: 'tournamentId = :tournament AND #region = :region AND round
 BETWEEN :start AND :end',
    ExpressionAttributeNames: {
        '#region': 'region'  // 'region' is a reserved keyword
    },
    ExpressionAttributeValues: {
        ':tournament': 'WINTER2024',
        ':region': 'NA-EAST',
        ':start': 'QUARTERFINALS',
        ':end': 'SEMIFINALS'
    }
}));
console.log(`  Found ${query2.Items.length} matches\n`);

// Query 3: Prefix matching with begins_with (treated as inequality)
console.log("Query 3: Matches in brackets starting with 'U'");
```

```
const query3 = await docClient.send(new QueryCommand({
    TableName: 'TournamentMatches',
    IndexName: 'TournamentRegionIndex',
    KeyConditionExpression: 'tournamentId = :tournament AND #region = :region AND round
 = :round AND begins_with(bracket, :prefix)',
    ExpressionAttributeNames: {
        '#region': 'region'  // 'region' is a reserved keyword
    },
    ExpressionAttributeValues: {
        ':tournament': 'WINTER2024',
        ':region': 'NA-EAST',
        ':round': 'SEMIFINALS',
        ':prefix': 'U'
    }
}));
console.log(`  Found ${query3.Items.length} matches\n`);

// Query 4: INVALID - condition after inequality
console.log("Query 4: Attempting condition after inequality (WILL FAIL)");
try {
    const query4 = await docClient.send(new QueryCommand({
        TableName: 'TournamentMatches',
        IndexName: 'TournamentRegionIndex',
        KeyConditionExpression: 'tournamentId = :tournament AND #region = :region AND
 round > :round AND bracket = :bracket',
        ExpressionAttributeNames: {
            '#region': 'region'  // 'region' is a reserved keyword
        },
        ExpressionAttributeValues: {
            ':tournament': 'WINTER2024',
            ':region': 'NA-EAST',
            ':round': 'QUARTERFINALS',
            ':bracket': 'UPPER'
        }
    }));
} catch (error) {
    console.log(`  Error: ${error.message}`);
    console.log(`  Cannot add conditions after inequality - it must be last\n`);
}
```

Inequality operator rules: You can use comparison operators (>, >=, <, <=), BETWEEN for range queries, and `begins_with()` for prefix matching. The inequality must be the last condition in your query.

Valid patterns:

- Equality conditions followed by inequality: `round = 'SEMIFINALS' AND bracket = 'UPPER' AND matchId > 'match-001'`
- Inequality on first attribute: `round BETWEEN 'QUARTERFINALS' AND 'SEMIFINALS'`
- Prefix matching as final condition: `round = 'SEMIFINALS' AND begins_with(bracket, 'U')`

Invalid patterns:

- Adding conditions after an inequality: `round > 'QUARTERFINALS' AND bracket = 'UPPER'`
- Using multiple inequalities: `round > 'QUARTERFINALS' AND bracket > 'L'`

> **Important**
>
> `begins_with()` is treated as an inequality condition, so no additional sort key conditions can follow it.

Step 6: Query PlayerMatchHistoryIndex Global Secondary Index with multi-attribute sort key

This example queries the PlayerMatchHistoryIndex which has a single partition key (`player1Id`) and a multi-attribute sort key (`matchDate` + `round`). This enables cross-tournament analysis by querying all matches for a specific player without knowing tournament IDs—whereas the base table would require separate queries per tournament-region combination.

Code example

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
import { DynamoDBDocumentClient, QueryCommand } from "@aws-sdk/lib-dynamodb";

const client = new DynamoDBClient({ region: 'us-west-2' });
const docClient = DynamoDBDocumentClient.from(client);

// Query 1: All matches for Player 101 across all tournaments
console.log("Query 1: All matches for Player 101");
const query1 = await docClient.send(new QueryCommand({
    TableName: 'TournamentMatches',
```

```
    IndexName: 'PlayerMatchHistoryIndex',
    KeyConditionExpression: 'player1Id = :player',
    ExpressionAttributeValues: {
        ':player': '101'
    }
}));

console.log(`  Found ${query1.Items.length} matches for Player 101:`);
query1.Items.forEach(match => {
    console.log(`    ${match.tournamentId}/${match.region} - ${match.matchDate} -
 ${match.round}`);
});
console.log();

// Query 2: Player 101 matches on specific date
console.log("Query 2: Player 101 matches on 2024-01-18");
const query2 = await docClient.send(new QueryCommand({
    TableName: 'TournamentMatches',
    IndexName: 'PlayerMatchHistoryIndex',
    KeyConditionExpression: 'player1Id = :player AND matchDate = :date',
    ExpressionAttributeValues: {
        ':player': '101',
        ':date': '2024-01-18'
    }
}));

console.log(`  Found ${query2.Items.length} matches\n`);

// Query 3: Player 101 SEMIFINALS matches on specific date
console.log("Query 3: Player 101 SEMIFINALS matches on 2024-01-18");
const query3 = await docClient.send(new QueryCommand({
    TableName: 'TournamentMatches',
    IndexName: 'PlayerMatchHistoryIndex',
    KeyConditionExpression: 'player1Id = :player AND matchDate = :date AND round
 = :round',
    ExpressionAttributeValues: {
        ':player': '101',
        ':date': '2024-01-18',
        ':round': 'SEMIFINALS'
    }
}));

console.log(`  Found ${query3.Items.length} matches\n`);
```

```
// Query 4: Player 101 matches in date range
console.log("Query 4: Player 101 matches in January 2024");
const query4 = await docClient.send(new QueryCommand({
    TableName: 'TournamentMatches',
    IndexName: 'PlayerMatchHistoryIndex',
    KeyConditionExpression: 'player1Id = :player AND matchDate BETWEEN :start
 AND :end',
    ExpressionAttributeValues: {
        ':player': '101',
        ':start': '2024-01-01',
        ':end': '2024-01-31'
    }
}));

console.log(`  Found ${query4.Items.length} matches\n`);
```

Pattern variations

Time-series data with multi-attribute keys

Optimize for time-series queries with hierarchical time attributes

Code example

```
{
    TableName: 'IoTReadings',
    // Base table: Simple partition key
    KeySchema: [
        { AttributeName: 'readingId', KeyType: 'HASH' }
    ],
    AttributeDefinitions: [
        { AttributeName: 'readingId', AttributeType: 'S' },
        { AttributeName: 'deviceId', AttributeType: 'S' },
        { AttributeName: 'locationId', AttributeType: 'S' },
        { AttributeName: 'year', AttributeType: 'S' },
        { AttributeName: 'month', AttributeType: 'S' },
        { AttributeName: 'day', AttributeType: 'S' },
        { AttributeName: 'timestamp', AttributeType: 'S' }
    ],
    // GSI with multi-attribute keys for time-series queries
    GlobalSecondaryIndexes: [{
        IndexName: 'DeviceLocationTimeIndex',
        KeySchema: [
            { AttributeName: 'deviceId', KeyType: 'HASH' },
```

```
            { AttributeName: 'locationId', KeyType: 'HASH' },
            { AttributeName: 'year', KeyType: 'RANGE' },
            { AttributeName: 'month', KeyType: 'RANGE' },
            { AttributeName: 'day', KeyType: 'RANGE' },
            { AttributeName: 'timestamp', KeyType: 'RANGE' }
        ],
        Projection: { ProjectionType: 'ALL' }
    }],
    BillingMode: 'PAY_PER_REQUEST'
}

// Query patterns enabled via GSI:
// - All readings for device in location
// - Readings for specific year
// - Readings for specific month in year
// - Readings for specific day
// - Readings in time range
```

Benefits: Natural time hierarchy (year → month → day → timestamp) enables efficient queries at any time granularity without date parsing or manipulation. Global Secondary Index automatically indexes all readings using their natural time attributes.

E-commerce orders with multi-attribute keys

Track orders with multiple dimensions

Code example

```
{
    TableName: 'Orders',
    // Base table: Simple partition key
    KeySchema: [
        { AttributeName: 'orderId', KeyType: 'HASH' }
    ],
    AttributeDefinitions: [
        { AttributeName: 'orderId', AttributeType: 'S' },
        { AttributeName: 'sellerId', AttributeType: 'S' },
        { AttributeName: 'region', AttributeType: 'S' },
        { AttributeName: 'orderDate', AttributeType: 'S' },
        { AttributeName: 'category', AttributeType: 'S' },
        { AttributeName: 'customerId', AttributeType: 'S' },
        { AttributeName: 'orderStatus', AttributeType: 'S' }
    ],
```

```
    GlobalSecondaryIndexes: [
        {
            IndexName: 'SellerRegionIndex',
            KeySchema: [
                { AttributeName: 'sellerId', KeyType: 'HASH' },
                { AttributeName: 'region', KeyType: 'HASH' },
                { AttributeName: 'orderDate', KeyType: 'RANGE' },
                { AttributeName: 'category', KeyType: 'RANGE' },
                { AttributeName: 'orderId', KeyType: 'RANGE' }
            ],
            Projection: { ProjectionType: 'ALL' }
        },
        {
            IndexName: 'CustomerOrdersIndex',
            KeySchema: [
                { AttributeName: 'customerId', KeyType: 'HASH' },
                { AttributeName: 'orderDate', KeyType: 'RANGE' },
                { AttributeName: 'orderStatus', KeyType: 'RANGE' }
            ],
            Projection: { ProjectionType: 'ALL' }
        }
    ],
    BillingMode: 'PAY_PER_REQUEST'
}

// SellerRegionIndex GSI queries:
// - Orders by seller and region
// - Orders by seller, region, and date
// - Orders by seller, region, date, and category

// CustomerOrdersIndex GSI queries:
// - Customer's orders
// - Customer's orders by date
// - Customer's orders by date and status
```

Hierarchical organization data

Model organizational hierarchies

Code example

```
{
    TableName: 'Employees',
    // Base table: Simple partition key
```

```
    KeySchema: [
        { AttributeName: 'employeeId', KeyType: 'HASH' }
    ],
    AttributeDefinitions: [
        { AttributeName: 'employeeId', AttributeType: 'S' },
        { AttributeName: 'companyId', AttributeType: 'S' },
        { AttributeName: 'divisionId', AttributeType: 'S' },
        { AttributeName: 'departmentId', AttributeType: 'S' },
        { AttributeName: 'teamId', AttributeType: 'S' },
        { AttributeName: 'skillCategory', AttributeType: 'S' },
        { AttributeName: 'skillLevel', AttributeType: 'S' },
        { AttributeName: 'yearsExperience', AttributeType: 'N' }
    ],
    GlobalSecondaryIndexes: [
        {
            IndexName: 'OrganizationIndex',
            KeySchema: [
                { AttributeName: 'companyId', KeyType: 'HASH' },
                { AttributeName: 'divisionId', KeyType: 'HASH' },
                { AttributeName: 'departmentId', KeyType: 'RANGE' },
                { AttributeName: 'teamId', KeyType: 'RANGE' },
                { AttributeName: 'employeeId', KeyType: 'RANGE' }
            ],
            Projection: { ProjectionType: 'ALL' }
        },
        {
            IndexName: 'SkillsIndex',
            KeySchema: [
                { AttributeName: 'skillCategory', KeyType: 'HASH' },
                { AttributeName: 'skillLevel', KeyType: 'RANGE' },
                { AttributeName: 'yearsExperience', KeyType: 'RANGE' }
            ],
            Projection: { ProjectionType: 'INCLUDE', NonKeyAttributes: ['employeeId',
 'name'] }
        }
    ],
    BillingMode: 'PAY_PER_REQUEST'
}

// OrganizationIndex GSI query patterns:
// - All employees in company/division
// - Employees in specific department
// - Employees in specific team
```

```
// SkillsIndex GSI query patterns:
// - Employees by skill and experience level
```

Sparse multi-attribute keys

Combine multi-attribute keys to make a sparse GSI

Code example

```
{
    TableName: 'Products',
    // Base table: Simple partition key
    KeySchema: [
        { AttributeName: 'productId', KeyType: 'HASH' }
    ],
    AttributeDefinitions: [
        { AttributeName: 'productId', AttributeType: 'S' },
        { AttributeName: 'categoryId', AttributeType: 'S' },
        { AttributeName: 'subcategoryId', AttributeType: 'S' },
        { AttributeName: 'averageRating', AttributeType: 'N' },
        { AttributeName: 'reviewCount', AttributeType: 'N' }
    ],
    GlobalSecondaryIndexes: [
        {
            IndexName: 'CategoryIndex',
            KeySchema: [
                { AttributeName: 'categoryId', KeyType: 'HASH' },
                { AttributeName: 'subcategoryId', KeyType: 'HASH' },
                { AttributeName: 'productId', KeyType: 'RANGE' }
            ],
            Projection: { ProjectionType: 'ALL' }
        },
        {
            IndexName: 'ReviewedProductsIndex',
            KeySchema: [
                { AttributeName: 'categoryId', KeyType: 'HASH' },
                { AttributeName: 'averageRating', KeyType: 'RANGE' },  // Optional
 attribute
                { AttributeName: 'reviewCount', KeyType: 'RANGE' }     // Optional
 attribute
            ],
            Projection: { ProjectionType: 'ALL' }
        }
    ],
```

```
    BillingMode: 'PAY_PER_REQUEST'
}

// Only products with reviews appear in ReviewedProductsIndex GSI
// Automatic filtering without application logic
// Multi-attribute sort key enables rating and count queries
```

SaaS multi-tenancy

Multi-tenant SaaS platform with customer isolation

Code example

```
// Table design
{
    TableName: 'SaasData',
    // Base table: Simple partition key
    KeySchema: [
        { AttributeName: 'resourceId', KeyType: 'HASH' }
    ],
    AttributeDefinitions: [
        { AttributeName: 'resourceId', AttributeType: 'S' },
        { AttributeName: 'tenantId', AttributeType: 'S' },
        { AttributeName: 'customerId', AttributeType: 'S' },
        { AttributeName: 'resourceType', AttributeType: 'S' }
    ],
    // GSI with multi-attribute keys for tenant-customer isolation
    GlobalSecondaryIndexes: [{
        IndexName: 'TenantCustomerIndex',
        KeySchema: [
            { AttributeName: 'tenantId', KeyType: 'HASH' },
            { AttributeName: 'customerId', KeyType: 'HASH' },
            { AttributeName: 'resourceType', KeyType: 'RANGE' },
            { AttributeName: 'resourceId', KeyType: 'RANGE' }
        ],
        Projection: { ProjectionType: 'ALL' }
    }],
    BillingMode: 'PAY_PER_REQUEST'
}

// Query GSI: All resources for tenant T001, customer C001
const resources = await docClient.send(new QueryCommand({
    TableName: 'SaasData',
    IndexName: 'TenantCustomerIndex',
```

```
    KeyConditionExpression: 'tenantId = :tenant AND customerId = :customer',
    ExpressionAttributeValues: {
        ':tenant': 'T001',
        ':customer': 'C001'
    }
}));

// Query GSI: Specific resource type for tenant/customer
const documents = await docClient.send(new QueryCommand({
    TableName: 'SaasData',
    IndexName: 'TenantCustomerIndex',
    KeyConditionExpression: 'tenantId = :tenant AND customerId = :customer AND
 resourceType = :type',
    ExpressionAttributeValues: {
        ':tenant': 'T001',
        ':customer': 'C001',
        ':type': 'document'
    }
}));
```

Benefits: Efficient queries within tenant-customer context and natural data organization.

Financial transactions

Banking system tracking account transactions using GSIs

Code example

```
// Table design
{
    TableName: 'BankTransactions',
    // Base table: Simple partition key
    KeySchema: [
        { AttributeName: 'transactionId', KeyType: 'HASH' }
    ],
    AttributeDefinitions: [
        { AttributeName: 'transactionId', AttributeType: 'S' },
        { AttributeName: 'accountId', AttributeType: 'S' },
        { AttributeName: 'year', AttributeType: 'S' },
        { AttributeName: 'month', AttributeType: 'S' },
        { AttributeName: 'day', AttributeType: 'S' },
        { AttributeName: 'transactionType', AttributeType: 'S' }
    ],
    GlobalSecondaryIndexes: [
```

```
        {
            IndexName: 'AccountTimeIndex',
            KeySchema: [
                { AttributeName: 'accountId', KeyType: 'HASH' },
                { AttributeName: 'year', KeyType: 'RANGE' },
                { AttributeName: 'month', KeyType: 'RANGE' },
                { AttributeName: 'day', KeyType: 'RANGE' },
                { AttributeName: 'transactionId', KeyType: 'RANGE' }
            ],
            Projection: { ProjectionType: 'ALL' }
        },
        {
            IndexName: 'TransactionTypeIndex',
            KeySchema: [
                { AttributeName: 'accountId', KeyType: 'HASH' },
                { AttributeName: 'transactionType', KeyType: 'RANGE' },
                { AttributeName: 'year', KeyType: 'RANGE' },
                { AttributeName: 'month', KeyType: 'RANGE' }
            ],
            Projection: { ProjectionType: 'ALL' }
        }
    ],
    BillingMode: 'PAY_PER_REQUEST'
}

// Query AccountTimeIndex GSI: All transactions for account in 2023
const yearTransactions = await docClient.send(new QueryCommand({
    TableName: 'BankTransactions',
    IndexName: 'AccountTimeIndex',
    KeyConditionExpression: 'accountId = :account AND #year = :year',
    ExpressionAttributeNames: { '#year': 'year' },
    ExpressionAttributeValues: {
        ':account': 'ACC-12345',
        ':year': '2023'
    }
}));

// Query AccountTimeIndex GSI: Transactions in specific month
const monthTransactions = await docClient.send(new QueryCommand({
    TableName: 'BankTransactions',
    IndexName: 'AccountTimeIndex',
    KeyConditionExpression: 'accountId = :account AND #year = :year AND #month
 = :month',
    ExpressionAttributeNames: { '#year': 'year', '#month': 'month' },
```

```
    ExpressionAttributeValues: {
        ':account': 'ACC-12345',
        ':year': '2023',
        ':month': '11'
    }
}));

// Query TransactionTypeIndex GSI: Deposits in 2023
const deposits = await docClient.send(new QueryCommand({
    TableName: 'BankTransactions',
    IndexName: 'TransactionTypeIndex',
    KeyConditionExpression: 'accountId = :account AND transactionType = :type AND #year
 = :year',
    ExpressionAttributeNames: { '#year': 'year' },
    ExpressionAttributeValues: {
        ':account': 'ACC-12345',
        ':type': 'deposit',
        ':year': '2023'
    }
}));
```

Complete example

The following example demonstrates multi-attribute keys from setup to cleanup:

Code example

```
import {
    DynamoDBClient,
    CreateTableCommand,
    DeleteTableCommand,
    waitUntilTableExists
} from "@aws-sdk/client-dynamodb";
import {
    DynamoDBDocumentClient,
    PutCommand,
    QueryCommand
} from "@aws-sdk/lib-dynamodb";

const client = new DynamoDBClient({ region: 'us-west-2' });
const docClient = DynamoDBDocumentClient.from(client);

async function multiAttributeKeysDemo() {
    console.log("Starting Multi-Attribute GSI Keys Demo\n");
```

```
    // Step 1: Create table with GSIs using multi-attribute keys
    console.log("1. Creating table with multi-attribute GSI keys...");
    await client.send(new CreateTableCommand({
        TableName: 'TournamentMatches',
        KeySchema: [
            { AttributeName: 'matchId', KeyType: 'HASH' }
        ],
        AttributeDefinitions: [
            { AttributeName: 'matchId', AttributeType: 'S' },
            { AttributeName: 'tournamentId', AttributeType: 'S' },
            { AttributeName: 'region', AttributeType: 'S' },
            { AttributeName: 'round', AttributeType: 'S' },
            { AttributeName: 'bracket', AttributeType: 'S' },
            { AttributeName: 'player1Id', AttributeType: 'S' },
            { AttributeName: 'matchDate', AttributeType: 'S' }
        ],
        GlobalSecondaryIndexes: [
            {
                IndexName: 'TournamentRegionIndex',
                KeySchema: [
                    { AttributeName: 'tournamentId', KeyType: 'HASH' },
                    { AttributeName: 'region', KeyType: 'HASH' },
                    { AttributeName: 'round', KeyType: 'RANGE' },
                    { AttributeName: 'bracket', KeyType: 'RANGE' },
                    { AttributeName: 'matchId', KeyType: 'RANGE' }
                ],
                Projection: { ProjectionType: 'ALL' }
            },
            {
                IndexName: 'PlayerMatchHistoryIndex',
                KeySchema: [
                    { AttributeName: 'player1Id', KeyType: 'HASH' },
                    { AttributeName: 'matchDate', KeyType: 'RANGE' },
                    { AttributeName: 'round', KeyType: 'RANGE' }
                ],
                Projection: { ProjectionType: 'ALL' }
            }
        ],
        BillingMode: 'PAY_PER_REQUEST'
    }));

    await waitUntilTableExists({ client, maxWaitTime: 120 }, { TableName:
 'TournamentMatches' });
```

```
    console.log("Table created\n");

    // Step 2: Insert tournament matches
    console.log("2. Inserting tournament matches...");
    const matches = [
        { matchId: 'match-001', tournamentId: 'WINTER2024', region: 'NA-EAST', round:
 'FINALS', bracket: 'CHAMPIONSHIP', player1Id: '101', player2Id: '103', matchDate:
 '2024-01-20', winner: '101', score: '3-1' },
        { matchId: 'match-002', tournamentId: 'WINTER2024', region: 'NA-EAST', round:
 'SEMIFINALS', bracket: 'UPPER', player1Id: '101', player2Id: '105', matchDate:
 '2024-01-18', winner: '101', score: '3-2' },
        { matchId: 'match-003', tournamentId: 'WINTER2024', region: 'NA-WEST', round:
 'FINALS', bracket: 'CHAMPIONSHIP', player1Id: '102', player2Id: '104', matchDate:
 '2024-01-20', winner: '102', score: '3-2' },
        { matchId: 'match-004', tournamentId: 'SPRING2024', region: 'NA-EAST', round:
 'QUARTERFINALS', bracket: 'UPPER', player1Id: '101', player2Id: '108', matchDate:
 '2024-03-15', winner: '101', score: '3-0' }
    ];

    for (const match of matches) {
        await docClient.send(new PutCommand({ TableName: 'TournamentMatches', Item:
 match }));
    }
    console.log(`Inserted ${matches.length} tournament matches\n`);

    // Step 3: Query GSI with multi-attribute partition key
    console.log("3. Query TournamentRegionIndex GSI: WINTER2024/NA-EAST matches");
    const gsiQuery1 = await docClient.send(new QueryCommand({
        TableName: 'TournamentMatches',
        IndexName: 'TournamentRegionIndex',
        KeyConditionExpression: 'tournamentId = :tournament AND #region = :region',
        ExpressionAttributeNames: { '#region': 'region' },
        ExpressionAttributeValues: { ':tournament': 'WINTER2024', ':region': 'NA-
EAST' }
    }));

    console.log(`  Found ${gsiQuery1.Items.length} matches:`);
    gsiQuery1.Items.forEach(match => {
        console.log(`    ${match.round} - ${match.bracket} - ${match.winner} won`);
    });

    // Step 4: Query GSI with multi-attribute sort key
    console.log("\n4. Query PlayerMatchHistoryIndex GSI: All matches for Player 101");
    const gsiQuery2 = await docClient.send(new QueryCommand({
```

```
        TableName: 'TournamentMatches',
        IndexName: 'PlayerMatchHistoryIndex',
        KeyConditionExpression: 'player1Id = :player',
        ExpressionAttributeValues: { ':player': '101' }
    }));

    console.log(`  Found ${gsiQuery2.Items.length} matches for Player 101:`);
    gsiQuery2.Items.forEach(match => {
        console.log(`    ${match.tournamentId}/${match.region} - ${match.matchDate} -
 ${match.round}`);
    });

    console.log("\nDemo complete");
    console.log("No synthetic keys needed - GSIs use native attributes automatically");
}

async function cleanup() {
    console.log("Deleting table...");
    await client.send(new DeleteTableCommand({ TableName: 'TournamentMatches' }));
    console.log("Table deleted");
}

// Run demo
multiAttributeKeysDemo().catch(console.error);

// Uncomment to cleanup:
// cleanup().catch(console.error);
```

Minimal code scaffold

Code example

```
// 1. Create table with GSI using multi-attribute keys
await client.send(new CreateTableCommand({
    TableName: 'MyTable',
    KeySchema: [
        { AttributeName: 'id', KeyType: 'HASH' }        // Simple base table PK
    ],
    AttributeDefinitions: [
        { AttributeName: 'id', AttributeType: 'S' },
        { AttributeName: 'attr1', AttributeType: 'S' },
        { AttributeName: 'attr2', AttributeType: 'S' },
        { AttributeName: 'attr3', AttributeType: 'S' },
        { AttributeName: 'attr4', AttributeType: 'S' }
```

```
    ],
    GlobalSecondaryIndexes: [{
        IndexName: 'MyGSI',
        KeySchema: [
            { AttributeName: 'attr1', KeyType: 'HASH' },    // GSI PK attribute 1
            { AttributeName: 'attr2', KeyType: 'HASH' },    // GSI PK attribute 2
            { AttributeName: 'attr3', KeyType: 'RANGE' },   // GSI SK attribute 1
            { AttributeName: 'attr4', KeyType: 'RANGE' }    // GSI SK attribute 2
        ],
        Projection: { ProjectionType: 'ALL' }
    }],
    BillingMode: 'PAY_PER_REQUEST'
}));

// 2. Insert items with native attributes (no concatenation needed for GSI)
await docClient.send(new PutCommand({
    TableName: 'MyTable',
    Item: {
        id: 'item-001',
        attr1: 'value1',
        attr2: 'value2',
        attr3: 'value3',
        attr4: 'value4',
        // ... other attributes
    }
}));

// 3. Query GSI with all partition key attributes
await docClient.send(new QueryCommand({
    TableName: 'MyTable',
    IndexName: 'MyGSI',
    KeyConditionExpression: 'attr1 = :v1 AND attr2 = :v2',
    ExpressionAttributeValues: {
        ':v1': 'value1',
        ':v2': 'value2'
    }
}));

// 4. Query GSI with sort key attributes (left-to-right)
await docClient.send(new QueryCommand({
    TableName: 'MyTable',
    IndexName: 'MyGSI',
    KeyConditionExpression: 'attr1 = :v1 AND attr2 = :v2 AND attr3 = :v3',
    ExpressionAttributeValues: {
```

```
        ':v1': 'value1',
        ':v2': 'value2',
        ':v3': 'value3'
    }
}));

// Note: If any attribute name is a DynamoDB reserved keyword, use
 ExpressionAttributeNames:
// KeyConditionExpression: 'attr1 = :v1 AND #attr2 = :v2'
// ExpressionAttributeNames: { '#attr2': 'attr2' }
```

Additional resources

- DynamoDB Best Practices
- Working with Tables and Data
- Global Secondary Indexes
- Query and Scan Operations

Managing Global Secondary Indexes in DynamoDB

This section describes how to create, modify, and delete global secondary indexes in Amazon DynamoDB.

Topics

- Creating a table with Global Secondary Indexes
- Describing the Global Secondary Indexes on a table
- Adding a Global Secondary Index to an existing table
- Deleting a Global Secondary Index
- Modifying a Global Secondary Index during creation

Creating a table with Global Secondary Indexes

To create a table with one or more global secondary indexes, use the `CreateTable` operation with the `GlobalSecondaryIndexes` parameter. For maximum query flexibility, you can create up to 20 global secondary indexes (default quota) per table.

You must specify one attribute to act as the index partition key. You can optionally specify another attribute for the index sort key. It is not necessary for either of these key attributes to be the

same as a key attribute in the table. For example, in the *GameScores* table (see Using Global Secondary Indexes in DynamoDB), neither `TopScore` nor `TopScoreDateTime` are key attributes. You could create a global secondary index with a partition key of `TopScore` and a sort key of `TopScoreDateTime`. You might use such an index to determine whether there is a correlation between high scores and the time of day a game is played.

Each index key attribute must be a scalar of type `String`, `Number`, or `Binary`. (It cannot be a document or a set.) You can project attributes of any data type into a global secondary index. This includes scalars, documents, and sets. For a complete list of data types, see Data types.

If using provisioned mode, you must provide `ProvisionedThroughput` settings for the index, consisting of `ReadCapacityUnits` and `WriteCapacityUnits`. These provisioned throughput settings are separate from those of the table, but behave in similar ways. For more information, see Provisioned throughput considerations for Global Secondary Indexes.

Global secondary indexes inherit the read/write capacity mode from the base table. For more information, see Considerations when switching capacity modes in DynamoDB.

Note

When creating a new GSI, it can be important to check if your choice of partition key is producing uneven or narrowed distribution of data or traffic across the new index's partition key values. If this occurs, you could be seeing backfill and write operations occurring at the same time and throttling writes to the base table. The service takes measures to minimize the potential for this scenario, but has no insight into the shape of customer data with respect to the index partition key, the chosen projection, or the sparseness of the index primary key.
If you suspect that your new global secondary index might have narrow or skewed data or traffic distribution across partition key values, consider the following before adding new indexes to operationally important tables.

- It might be safest to add the index at a time when your application is driving the least amount of traffic.
- Consider enabling CloudWatch Contributor Insights on your base table and indexes. This will give you valuable insight into your traffic distribution.
- Watch `WriteThrottleEvents`, `ThrottledRequests`, and `OnlineIndexPercentageProgress` CloudWatch metrics throughout the process. Adjust the provisioned write capacity as required to complete the backfill in a

reasonable time without any significant throttling effects on your ongoing operations. `OnlineIndexConsumedWriteCapacity` and `OnlineThrottleEvents` are expected to show 0 during index backfill.

- Be prepared to cancel the index creation if you experience operational impact due to write throttling.

Describing the Global Secondary Indexes on a table

To view the status of all the global secondary indexes on a table, use the `DescribeTable` operation. The `GlobalSecondaryIndexes` portion of the response shows all of the indexes on the table, along with the current status of each (`IndexStatus`).

The `IndexStatus` for a global secondary index will be one of the following:

- `CREATING` — The index is currently being created, and is not yet available for use.
- `ACTIVE` — The index is ready for use, and applications can perform `Query` operations on the index.
- `UPDATING` — The provisioned throughput settings of the index are being changed.
- `DELETING` — The index is currently being deleted, and can no longer be used.

When DynamoDB has finished building a global secondary index, the index status changes from `CREATING` to `ACTIVE`.

Adding a Global Secondary Index to an existing table

To add a global secondary index to an existing table, use the `UpdateTable` operation with the `GlobalSecondaryIndexUpdates` parameter. You must provide the following:

- An index name. The name must be unique among all the indexes on the table.
- The key schema of the index. You must specify one attribute for the index partition key; you can optionally specify another attribute for the index sort key. It is not necessary for either of these key attributes to be the same as a key attribute in the table. The data types for each schema attribute must be scalar: `String`, `Number`, or `Binary`.
- The attributes to be projected from the table into the index:
  - `KEYS_ONLY` — Each item in the index consists only of the table partition key and sort key values, plus the index key values.

- `INCLUDE` — In addition to the attributes described in `KEYS_ONLY`, the secondary index includes other non-key attributes that you specify.
- `ALL` — The index includes all of the attributes from the source table.

- The provisioned throughput settings for the index, consisting of `ReadCapacityUnits` and `WriteCapacityUnits`. These provisioned throughput settings are separate from those of the table.

You can only create one global secondary index per `UpdateTable` operation.

Phases of index creation

When you add a new global secondary index to an existing table, the table continues to be available while the index is being built. However, the new index is not available for Query operations until its status changes from `CREATING` to `ACTIVE`.

> **Note**
>
> Global secondary index creation does not use Application Auto Scaling. Increasing the `MIN` Application Auto Scaling capacity will not decrease the creation time of the global secondary index.

Behind the scenes, DynamoDB builds the index in two phases:

Resource Allocation

DynamoDB allocates the compute and storage resources that are needed for building the index.

During the resource allocation phase, the `IndexStatus` attribute is `CREATING` and the `Backfilling` attribute is false. Use the `DescribeTable` operation to retrieve the status of a table and all of its secondary indexes.

While the index is in the resource allocation phase, you can't delete the index or delete its parent table. You also can't modify the provisioned throughput of the index or the table. You cannot add or delete other indexes on the table. However, you can modify the provisioned throughput of these other indexes.

Backfilling

For each item in the table, DynamoDB determines which set of attributes to write to the index based on its projection (`KEYS_ONLY`, `INCLUDE`, or `ALL`). It then writes these attributes to the index. During the backfill phase, DynamoDB tracks the items that are being added, deleted, or updated in the table. The attributes from these items are also added, deleted, or updated in the index as appropriate.

During the backfilling phase, the `IndexStatus` attribute is set to `CREATING`, and the `Backfilling` attribute is true. Use the `DescribeTable` operation to retrieve the status of a table and all of its secondary indexes.

While the index is backfilling, you cannot delete its parent table. However, you can still delete the index or modify the provisioned throughput of the table and any of its global secondary indexes.

> **Note**
>
> During the backfilling phase, some writes of violating items might succeed while others are rejected. After backfilling, all writes to items that violate the new index's key schema are rejected. We recommend that you run the Violation Detector tool after the backfill phase finishes to detect and resolve any key violations that might have occurred. For more information, see Detecting and correcting index key violations in DynamoDB.

While the resource allocation and backfilling phases are in progress, the index is in the `CREATING` state. During this time, DynamoDB performs read operations on the table. You are not charged for read operations from the base table to populate the global secondary index.

When the index build is complete, its status changes to `ACTIVE`. You can't `Query` or `Scan` the index until it is `ACTIVE`.

> **Note**
>
> In some cases, DynamoDB can't write data from the table to the index because of index key violations. This can occur if:
>
> - The data type of an attribute value does not match the data type of an index key schema data type.

- The size of an attribute exceeds the maximum length for an index key attribute.
- An index key attribute has an empty String or empty Binary attribute value.

Index key violations do not interfere with global secondary index creation. However, when the index becomes `ACTIVE`, the violating keys are not present in the index. DynamoDB provides a standalone tool for finding and resolving these issues. For more information, see Detecting and correcting index key violations in DynamoDB.

Adding a Global Secondary Index to a large table

The time required for building a global secondary index depends on several factors, such as the following:

- The size of the table
- The number of items in the table that qualify for inclusion in the index
- The number of attributes projected into the index
- Write activity on the main table during index builds

If you are adding a global secondary index to a very large table, it might take a long time for the creation process to complete. To monitor progress and determine whether the index has sufficient write capacity, consult the following Amazon CloudWatch metrics:

- `OnlineIndexPercentageProgress`

For more information about CloudWatch metrics related to DynamoDB, see DynamoDB metrics.

Important

You may need to allowlist very large tables before creating or updating a Global Secondary Index. Please reach out to AWS Support to allowlist your tables.

While an index is being backfilled, DynamoDB uses internal system capacity to read from the table. This is to minimize the impact of the index creation and to assure that your table does not run out of read capacity.

Deleting a Global Secondary Index

If you no longer need a global secondary index, you can delete it using the `UpdateTable` operation.

You can delete only one global secondary index per `UpdateTable` operation.

While the global secondary index is being deleted, there is no effect on any read or write activity in the parent table. While the deletion is in progress, you can still modify the provisioned throughput on other indexes.

> **Note**
>
> - When you delete a table using the `DeleteTable` action, all of the global secondary indexes on that table are also deleted.
> - Your account will not be charged for the delete operation of the global secondary index.

Modifying a Global Secondary Index during creation

While an index is being built, you can use the `DescribeTable` operation to determine what phase it is in. The description for the index includes a Boolean attribute, `Backfilling`, to indicate whether DynamoDB is currently loading the index with items from the table. If `Backfilling` is true, the resource allocation phase is complete and the index is now backfilling.

During the backfilling phase, you can delete the index that is being created. During this phase, you can't add or delete other indexes on the table.

> **Note**
>
> For indexes that were created as part of a `CreateTable` operation, the `Backfilling` attribute does not appear in the `DescribeTable` output. For more information, see Phases of index creation.

Detecting and correcting index key violations in DynamoDB

During the backfill phase of global secondary index creation, Amazon DynamoDB examines each item in the table to determine whether it is eligible for inclusion in the index. Some items might

not be eligible because they would cause index key violations. In these cases, the items remain in the table, but the index doesn't have a corresponding entry for that item.

An *index key violation* occurs in the following situations:

- There is a data type mismatch between an attribute value and the index key schema data type. For example, suppose that one of the items in the `GameScores` table had a `TopScore` value of type `String`. If you added a global secondary index with a partition key of `TopScore`, of type `Number`, the item from the table would violate the index key.
- An attribute value from the table exceeds the maximum length for an index key attribute. The maximum length of a partition key is 2048 bytes, and the maximum length of a sort key is 1024 bytes. If any of the corresponding attribute values in the table exceed these limits, the item from the table would violate the index key.

> **Note**
>
> If a String or Binary attribute value is set for an attribute that is used as an index key, then the attribute value must have a length greater than zero;, otherwise, the item from the table would violate the index key.
> This tool does not flag this index key violation, at this time.

If an index key violation occurs, the backfill phase continues without interruption. However, any violating items are not included in the index. After the backfill phase completes, all writes to items that violate the new index's key schema will be rejected.

To identify and fix attribute values in a table that violate an index key, use the Violation Detector tool. To run Violation Detector, you create a configuration file that specifies the name of a table to be scanned, the names and data types of the global secondary index partition key and sort key, and what actions to take if any index key violations are found. Violation Detector can run in one of two different modes:

- **Detection mode** — Detect index key violations. Use detection mode to report the items in the table that would cause key violations in a global secondary index. (You can optionally request that these violating table items be deleted immediately when they are found.) The output from detection mode is written to a file, which you can use for further analysis.

- **Correction mode** — Correct index key violations. In correction mode, Violation Detector reads an input file with the same format as the output file from detection mode. Correction mode reads the records from the input file and, for each record, it either deletes or updates the corresponding items in the table. (Note that if you choose to update the items, you must edit the input file and set appropriate values for these updates.)

Downloading and running Violation Detector

Violation Detector is available as an executable Java Archive (`.jar` file), and runs on Windows, macOS, or Linux computers. Violation Detector requires Java 1.7 (or later) and Apache Maven.

- Download violation detector from GitHub

Follow the instructions in the `README.md` file to download and install Violation Detector using Maven.

To start Violation Detector, go to the directory where you have built `ViolationDetector.java` and enter the following command.

```
java -jar ViolationDetector.jar [options]
```

The Violation Detector command line accepts the following options:

- `-h | --help` — Prints a usage summary and options for Violation Detector.
- `-p | --configFilePath value` — The fully qualified name of a Violation Detector configuration file. For more information, see The Violation Detector configuration file.
- `-t | --detect value` — Detect index key violations in the table, and write them to the Violation Detector output file. If the value of this parameter is set to `keep`, items with key violations are not modified. If the value is set to `delete`, items with key violations are deleted from the table.
- `-c | --correct value` — Read index key violations from an input file, and take corrective actions on the items in the table. If the value of this parameter is set to `update`, items with key violations are updated with new, non-violating values. If the value is set to `delete`, items with key violations are deleted from the table.

The Violation Detector configuration file

At runtime, the Violation Detector tool requires a configuration file. The parameters in this file determine which DynamoDB resources that Violation Detector can access, and how much provisioned throughput it can consume. The following table describes these parameters.

Parameter name	Description	Required?

`awsCredentialsFile`	The fully qualified name of a file containing your AWS credentials. The credentials file must be in the following format:
	`accessKey = access_key_id_goes_here`
	`secretKey = secret_key_goes_here`
		Yes

`dynamoDBRegion`	The AWS Region in which the table resides. For example: `us-west-2` .	Yes
`tableName`	The name of the DynamoDB table to be scanned.	Yes
`gsiHashKeyName`	The name of the index partition key.	Yes
`gsiHashKeyType`	The data type of the index partition key—`String`, `Number`, or `Binary`: `S | N | B`	Yes
`gsiRangeKeyName`	The name of the index sort key. Do not specify this parameter if the index only	No

Parameter name	Description	Required?
	has a simple primary key (partition key).	
`gsiRangeKeyType`	The data type of the index sort key—`String`, `Number`, or `Binary`: `S | N | B` Do not specify this parameter if the index only has a simple primary key (partition key).	No
`recordDetails`	Whether to write the full details of index key violations to the output file. If set to `true` (the default), full information about the violating items is reported. If set to `false`, only the number of violations is reported.	No
`recordGsiValueInViolationRecord`	Whether to write the values of the violating index keys to the output file. If set to `true` (default), the key values are reported. If set to `false`, the key values are not reported.	No

Parameter name	Description	Required?

`detectionOutputPath`	The full path of the Violation Detector output file. This parameter supports writing to a local directory or to Amazon Simple Storage Service (Amazon S3). The following are examples:
	
	`detectionOutputPath = //local/path/filename.csv`
	
	`detectionOutputPath = s3://bucket/filename.csv`
	
	Information in the output file appears in comma-separated values (CSV) format. If you don't set `detectionOutputPath`, the output file is named `violation_detection.csv` and is written to your current working directory.
		No

Parameter name	Description	Required?
`numOfSegments`	The number of parallel scan segments to be used when Violation Detector scans the table. The default value is 1, meaning that the table is scanned in a sequential manner. If the value is 2 or higher, then Violation Detector divides the table into that many logical segments and an equal number of scan threads. The maximum setting for `numOfSegments` is 4096. For larger tables, a parallel scan is generally faster than a sequential scan. In addition, if the table is large enough to span multiple partitions, a parallel scan distributes its read activity evenly across multiple partitions. For more information about parallel scans in DynamoDB, see Parallel scan.	No

Parameter name	Description	Required?
`numOfViolations`	The upper limit of index key violations to write to the output file. If set to `-1` (the default), the entire table is scanned. If set to a positive integer, then Violation Detector stops after it encounters that number of violations.	No
`numOfRecords`	The number of items in the table to be scanned. If set to -1 (the default), the entire table is scanned. If set to a positive integer, Violation Detector stops after it scans that many items in the table.	No
`readWriteIOPSPercent`	Regulates the percentage of provisioned read capacity units that are consumed during the table scan. Valid values range from 1 to `100`. The default value (25) means that Violation Detector will consume no more than 25% of the table's provisioned read throughput.	No

Parameter name	Description	Required?
`correctionInputPath`	The full path of the Violation Detector correction input file. If you run Violation Detector in correction mode, the contents of this file are used to modify or delete data items in the table that violate the global secondary index. The format of the `correctionInputPath` file is the same as that of the `detectionOutputPath` file. This lets you process the output from detection mode as input in correction mode.	No

Parameter name	Description	Required?

`correctionOutputPath`	The full path of the Violation Detector correction output file. This file is created only if there are update errors.
	
	This parameter supports writing to a local directory or to Amazon S3. The following are examples:
	
	`correctionOutputPath = //local/path/filename.csv`
	
	`correctionOutputPath = s3://bucket/filename.csv`
	
	Information in the output file appears in CSV format. If you don't set `correctionOutputPath`, the output file is named `violation_update_errors.csv` and is written to your current working directory.
		No

Detection

To detect index key violations, use Violation Detector with the `--detect` command line option. To show how this option works, consider the `ProductCatalog` table. The following is a list of items in the table. Only the primary key (`Id`) and the `Price` attribute are shown.

Id (primary key)	Price
101	5
102	20
103	200
201	100
202	200
203	300
204	400
205	500

All of the values for `Price` are of type `Number`. However, because DynamoDB is schemaless, it is possible to add an item with a non-numeric `Price`. For example, suppose that you add another item to the `ProductCatalog` table.

Id (primary key)	Price
999	`"Hello"`

The table now has a total of nine items.

Now you add a new global secondary index to the table: `PriceIndex`. The primary key for this index is a partition key, `Price`, which is of type `Number`. After the index has been built, it will contain eight items—but the `ProductCatalog` table has nine items. The reason for this discrepancy is that the value `"Hello"` is of type `String`, but `PriceIndex` has a primary key of type `Number`. The `String` value violates the global secondary index key, so it is not present in the index.

To use Violation Detector in this scenario, you first create a configuration file such as the following.

```
# Properties file for violation detection tool configuration.
# Parameters that are not specified will use default values.

awsCredentialsFile = /home/alice/credentials.txt
dynamoDBRegion = us-west-2
tableName = ProductCatalog
gsiHashKeyName = Price
gsiHashKeyType = N
recordDetails = true
recordGsiValueInViolationRecord = true
detectionOutputPath = ./gsi_violation_check.csv
correctionInputPath = ./gsi_violation_check.csv
numOfSegments = 1
readWriteIOPSPercent = 40
```

Next, you run Violation Detector as in the following example.

```
$  java -jar ViolationDetector.jar --configFilePath config.txt --detect keep

Violation detection started: sequential scan, Table name: ProductCatalog, GSI name:
 PriceIndex
Progress: Items scanned in total: 9,    Items scanned by this thread: 9,    Violations
 found by this thread: 1, Violations deleted by this thread: 0
Violation detection finished: Records scanned: 9, Violations found: 1, Violations
 deleted: 0, see results at: ./gsi_violation_check.csv
```

If the `recordDetails` config parameter is set to `true`, Violation Detector writes details of each violation to the output file, as in the following example.

```
Table Hash Key,GSI Hash Key Value,GSI Hash Key Violation Type,GSI Hash Key Violation
 Description,GSI Hash Key Update Value(FOR USER),Delete Blank Attributes When Updating?
(Y/N)

999,"{""S"":""Hello""}",Type Violation,Expected: N Found: S,,
```

The output file is in CSV format. The first line in the file is a header, followed by one record per item that violates the index key. The fields of these violation records are as follows:

- **Table hash key** — The partition key value of the item in the table.
- **Table range key** — The sort key value of the item in the table.
- **GSI hash key value** — The partition key value of the global secondary index.

- **GSI hash key violation type** — Either `Type Violation` or `Size Violation`.
- **GSI hash key violation description** — The cause of the violation.
- **GSI hash key update Value(FOR USER)** — In correction mode, a new user-supplied value for the attribute.
- **GSI range key value** — The sort key value of the global secondary index.
- **GSI range key violation type** — Either `Type Violation` or `Size Violation`.
- **GSI range key violation description** — The cause of the violation.
- **GSI range key update Value(FOR USER)** — In correction mode, a new user-supplied value for the attribute.
- **Delete blank attribute when Updating(Y/N)** — In correction mode, determines whether to delete (Y) or keep (N) the violating item in the table—but only if either of the following fields are blank:
  - `GSI Hash Key Update Value(FOR USER)`
  - `GSI Range Key Update Value(FOR USER)`

  If either of these fields are non-blank, then `Delete Blank Attribute When Updating(Y/N)` has no effect.

> **Note**
>
> The output format might vary, depending on the configuration file and command line options. For example, if the table has a simple primary key (without a sort key), no sort key fields will be present in the output.
> The violation records in the file might not be in sorted order.

Correction

To correct index key violations, use Violation Detector with the `--correct` command line option. In correction mode, Violation Detector reads the input file specified by the `correctionInputPath` parameter. This file has the same format as the `detectionOutputPath` file, so that you can use the output from detection as input for correction.

Violation Detector provides two different ways to correct index key violations:

- **Delete violations** — Delete the table items that have violating attribute values.

- **Update violations** — Update the table items, replacing the violating attributes with non-violating values.

In either case, you can use the output file from detection mode as input for correction mode.

Continuing with the `ProductCatalog` example, suppose that you want to delete the violating item from the table. To do this, you use the following command line.

```
$  java -jar ViolationDetector.jar --configFilePath config.txt --correct delete
```

At this point, you are asked to confirm whether you want to delete the violating items.

```
Are you sure to delete all violations on the table?y/n
y
Confirmed, will delete violations on the table...
Violation correction from file started: Reading records from file: ./
gsi_violation_check.csv, will delete these records from table.
Violation correction from file finished: Violations delete: 1, Violations Update: 0
```

Now both `ProductCatalog` and `PriceIndex` have the same number of items.

Working with Global Secondary Indexes: Java

You can use the AWS SDK for Java Document API to create an Amazon DynamoDB table with one or more global secondary indexes, describe the indexes on the table, and perform queries using the indexes.

The following are the common steps for table operations.

1. Create an instance of the `DynamoDB` class.
2. Provide the required and optional parameters for the operation by creating the corresponding request objects.
3. Call the appropriate method provided by the client that you created in the preceding step.

Topics

- Create a table with a Global Secondary Index
- Describe a table with a Global Secondary Index
- Query a Global Secondary Index

- Example: Global Secondary Indexes using the AWS SDK for Java document API

Create a table with a Global Secondary Index

You can create global secondary indexes at the same time that you create a table. To do this, use `CreateTable` and provide your specifications for one or more global secondary indexes. The following Java code example creates a table to hold information about weather data. The partition key is `Location` and the sort key is `Date`. A global secondary index named `PrecipIndex` allows fast access to precipitation data for various locations.

The following are the steps to create a table with a global secondary index, using the DynamoDB document API.

1. Create an instance of the `DynamoDB` class.
2. Create an instance of the `CreateTableRequest` class to provide the request information.

   You must provide the table name, its primary key, and the provisioned throughput values. For the global secondary index, you must provide the index name, its provisioned throughput settings, the attribute definitions for the index sort key, the key schema for the index, and the attribute projection.
3. Call the `createTable` method by providing the request object as a parameter.

The following Java code example demonstrates the preceding steps. The code creates a table (`WeatherData`) with a global secondary index (`PrecipIndex`). The index partition key is `Date` and its sort key is `Precipitation`. All of the table attributes are projected into the index. Users can query this index to obtain weather data for a particular date, optionally sorting the data by precipitation amount.

Because `Precipitation` is not a key attribute for the table, it is not required. However, `WeatherData` items without `Precipitation` do not appear in `PrecipIndex`.

```
AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();
DynamoDB dynamoDB = new DynamoDB(client);

// Attribute definitions
ArrayList<AttributeDefinition> attributeDefinitions = new
 ArrayList<AttributeDefinition>();

attributeDefinitions.add(new AttributeDefinition()
```

```
    .withAttributeName("Location")
    .withAttributeType("S"));
attributeDefinitions.add(new AttributeDefinition()
    .withAttributeName("Date")
    .withAttributeType("S"));
attributeDefinitions.add(new AttributeDefinition()
    .withAttributeName("Precipitation")
    .withAttributeType("N"));

// Table key schema
ArrayList<KeySchemaElement> tableKeySchema = new ArrayList<KeySchemaElement>();
tableKeySchema.add(new KeySchemaElement()
    .withAttributeName("Location")
    .withKeyType(KeyType.HASH));  //Partition key
tableKeySchema.add(new KeySchemaElement()
    .withAttributeName("Date")
    .withKeyType(KeyType.RANGE));  //Sort key

// PrecipIndex
GlobalSecondaryIndex precipIndex = new GlobalSecondaryIndex()
    .withIndexName("PrecipIndex")
    .withProvisionedThroughput(new ProvisionedThroughput()
        .withReadCapacityUnits((long) 10)
        .withWriteCapacityUnits((long) 1))
        .withProjection(new Projection().withProjectionType(ProjectionType.ALL));

ArrayList<KeySchemaElement> indexKeySchema = new ArrayList<KeySchemaElement>();

indexKeySchema.add(new KeySchemaElement()
    .withAttributeName("Date")
    .withKeyType(KeyType.HASH));  //Partition key
indexKeySchema.add(new KeySchemaElement()
    .withAttributeName("Precipitation")
    .withKeyType(KeyType.RANGE));  //Sort key

precipIndex.setKeySchema(indexKeySchema);

CreateTableRequest createTableRequest = new CreateTableRequest()
    .withTableName("WeatherData")
    .withProvisionedThroughput(new ProvisionedThroughput()
        .withReadCapacityUnits((long) 5)
        .withWriteCapacityUnits((long) 1))
    .withAttributeDefinitions(attributeDefinitions)
    .withKeySchema(tableKeySchema)
```

```
    .withGlobalSecondaryIndexes(precipIndex);

Table table = dynamoDB.createTable(createTableRequest);
System.out.println(table.getDescription());
```

You must wait until DynamoDB creates the table and sets the table status to `ACTIVE`. After that, you can begin putting data items into the table.

Describe a table with a Global Secondary Index

To get information about global secondary indexes on a table, use `DescribeTable`. For each index, you can access its name, key schema, and projected attributes.

The following are the steps to access global secondary index information a table.

1. Create an instance of the `DynamoDB` class.
2. Create an instance of the `Table` class to represent the index you want to work with.
3. Call the `describe` method on the `Table` object.

The following Java code example demonstrates the preceding steps.

Example

```
AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();
DynamoDB dynamoDB = new DynamoDB(client);

Table table = dynamoDB.getTable("WeatherData");
TableDescription tableDesc = table.describe();


Iterator<GlobalSecondaryIndexDescription> gsiIter =
 tableDesc.getGlobalSecondaryIndexes().iterator();
while (gsiIter.hasNext()) {
    GlobalSecondaryIndexDescription gsiDesc = gsiIter.next();
    System.out.println("Info for index "
         + gsiDesc.getIndexName() + ":");

    Iterator<KeySchemaElement> kseIter = gsiDesc.getKeySchema().iterator();
    while (kseIter.hasNext()) {
        KeySchemaElement kse = kseIter.next();
        System.out.printf("\t%s: %s\n", kse.getAttributeName(), kse.getKeyType());
    }
```

```
    Projection projection = gsiDesc.getProjection();
    System.out.println("\tThe projection type is: "
        + projection.getProjectionType());
    if (projection.getProjectionType().toString().equals("INCLUDE")) {
        System.out.println("\t\tThe non-key projected attributes are: "
            + projection.getNonKeyAttributes());
    }
}
```

Query a Global Secondary Index

You can use `Query` on a global secondary index, in much the same way you `Query` a table. You need to specify the index name, the query criteria for the index partition key and sort key (if present), and the attributes that you want to return. In this example, the index is `PrecipIndex`, which has a partition key of `Date` and a sort key of `Precipitation`. The index query returns all of the weather data for a particular date, where the precipitation is greater than zero.

The following are the steps to query a global secondary index using the AWS SDK for Java Document API.

1. Create an instance of the `DynamoDB` class.
2. Create an instance of the `Table` class to represent the index you want to work with.
3. Create an instance of the `Index` class for the index you want to query.
4. Call the `query` method on the `Index` object.

The attribute name `Date` is a DynamoDB reserved word. Therefore, you must use an expression attribute name as a placeholder in the `KeyConditionExpression`.

The following Java code example demonstrates the preceding steps.

Example

```
AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();
DynamoDB dynamoDB = new DynamoDB(client);

Table table = dynamoDB.getTable("WeatherData");
Index index = table.getIndex("PrecipIndex");

QuerySpec spec = new QuerySpec()
    .withKeyConditionExpression("#d = :v_date and Precipitation = :v_precip")
    .withNameMap(new NameMap()
```

```
        .with("#d", "Date"))
    .withValueMap(new ValueMap()
        .withString(":v_date","2013-08-10")
        .withNumber(":v_precip",0));

ItemCollection<QueryOutcome> items = index.query(spec);
Iterator<Item> iter = items.iterator();
while (iter.hasNext()) {
    System.out.println(iter.next().toJSONPretty());
}
```

Example: Global Secondary Indexes using the AWS SDK for Java document API

The following Java code example shows how to work with global secondary indexes. The example creates a table named `Issues`, which might be used in a simple bug tracking system for software development. The partition key is `IssueId` and the sort key is `Title`. There are three global secondary indexes on this table:

- `CreateDateIndex` — The partition key is `CreateDate` and the sort key is `IssueId`. In addition to the table keys, the attributes `Description` and `Status` are projected into the index.
- `TitleIndex` — The partition key is `Title` and the sort key is `IssueId`. No attributes other than the table keys are projected into the index.
- `DueDateIndex` — The partition key is `DueDate`, and there is no sort key. All of the table attributes are projected into the index.

After the `Issues` table is created, the program loads the table with data representing software bug reports. It then queries the data using the global secondary indexes. Finally, the program deletes the `Issues` table.

For step-by-step instructions for testing the following example, see Java code examples.

Example

```
package com.amazonaws.codesamples.document;

import java.util.ArrayList;
import java.util.Iterator;

import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
```

```
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;
import com.amazonaws.services.dynamodbv2.document.DynamoDB;
import com.amazonaws.services.dynamodbv2.document.Index;
import com.amazonaws.services.dynamodbv2.document.Item;
import com.amazonaws.services.dynamodbv2.document.ItemCollection;
import com.amazonaws.services.dynamodbv2.document.QueryOutcome;
import com.amazonaws.services.dynamodbv2.document.Table;
import com.amazonaws.services.dynamodbv2.document.spec.QuerySpec;
import com.amazonaws.services.dynamodbv2.document.utils.ValueMap;
import com.amazonaws.services.dynamodbv2.model.AttributeDefinition;
import com.amazonaws.services.dynamodbv2.model.CreateTableRequest;
import com.amazonaws.services.dynamodbv2.model.GlobalSecondaryIndex;
import com.amazonaws.services.dynamodbv2.model.KeySchemaElement;
import com.amazonaws.services.dynamodbv2.model.KeyType;
import com.amazonaws.services.dynamodbv2.model.Projection;
import com.amazonaws.services.dynamodbv2.model.ProvisionedThroughput;

public class DocumentAPIGlobalSecondaryIndexExample {

    static AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();
    static DynamoDB dynamoDB = new DynamoDB(client);

    public static String tableName = "Issues";

    public static void main(String[] args) throws Exception {

        createTable();
        loadData();

        queryIndex("CreateDateIndex");
        queryIndex("TitleIndex");
        queryIndex("DueDateIndex");

        deleteTable(tableName);

    }

    public static void createTable() {

        // Attribute definitions
        ArrayList<AttributeDefinition> attributeDefinitions = new
 ArrayList<AttributeDefinition>();
```

```
        attributeDefinitions.add(new
AttributeDefinition().withAttributeName("IssueId").withAttributeType("S"));
        attributeDefinitions.add(new
AttributeDefinition().withAttributeName("Title").withAttributeType("S"));
        attributeDefinitions.add(new
AttributeDefinition().withAttributeName("CreateDate").withAttributeType("S"));
        attributeDefinitions.add(new
AttributeDefinition().withAttributeName("DueDate").withAttributeType("S"));

        // Key schema for table
        ArrayList<KeySchemaElement> tableKeySchema = new ArrayList<KeySchemaElement>();
        tableKeySchema.add(new
KeySchemaElement().withAttributeName("IssueId").withKeyType(KeyType.HASH)); //
Partition

                    // key
        tableKeySchema.add(new
KeySchemaElement().withAttributeName("Title").withKeyType(KeyType.RANGE)); // Sort

                   // key

        // Initial provisioned throughput settings for the indexes
        ProvisionedThroughput ptIndex = new
ProvisionedThroughput().withReadCapacityUnits(1L)
                .withWriteCapacityUnits(1L);

        // CreateDateIndex
        GlobalSecondaryIndex createDateIndex = new
GlobalSecondaryIndex().withIndexName("CreateDateIndex")
                .withProvisionedThroughput(ptIndex)
                .withKeySchema(new
KeySchemaElement().withAttributeName("CreateDate").withKeyType(KeyType.HASH), //
Partition

                          // key
                        new
KeySchemaElement().withAttributeName("IssueId").withKeyType(KeyType.RANGE)) // Sort

                 // key
                .withProjection(
                        new
Projection().withProjectionType("INCLUDE").withNonKeyAttributes("Description",
"Status"));
```

```
        // TitleIndex
        GlobalSecondaryIndex titleIndex = new
GlobalSecondaryIndex().withIndexName("TitleIndex")
                .withProvisionedThroughput(ptIndex)
                .withKeySchema(new
KeySchemaElement().withAttributeName("Title").withKeyType(KeyType.HASH), // Partition

                     // key
                        new
KeySchemaElement().withAttributeName("IssueId").withKeyType(KeyType.RANGE)) // Sort

                 // key
                .withProjection(new Projection().withProjectionType("KEYS_ONLY"));

        // DueDateIndex
        GlobalSecondaryIndex dueDateIndex = new
GlobalSecondaryIndex().withIndexName("DueDateIndex")
                .withProvisionedThroughput(ptIndex)
                .withKeySchema(new
KeySchemaElement().withAttributeName("DueDate").withKeyType(KeyType.HASH)) //
Partition

                       // key
                .withProjection(new Projection().withProjectionType("ALL"));

        CreateTableRequest createTableRequest = new
CreateTableRequest().withTableName(tableName)
                .withProvisionedThroughput(
                        new ProvisionedThroughput().withReadCapacityUnits((long)
1).withWriteCapacityUnits((long) 1))

.withAttributeDefinitions(attributeDefinitions).withKeySchema(tableKeySchema)
                .withGlobalSecondaryIndexes(createDateIndex, titleIndex, dueDateIndex);

        System.out.println("Creating table " + tableName + "...");
        dynamoDB.createTable(createTableRequest);

        // Wait for table to become active
        System.out.println("Waiting for " + tableName + " to become ACTIVE...");
        try {
            Table table = dynamoDB.getTable(tableName);
            table.waitForActive();
        } catch (InterruptedException e) {
            e.printStackTrace();
```

```
        }
    }

    public static void queryIndex(String indexName) {

        Table table = dynamoDB.getTable(tableName);


System.out.println("\n***********************************************************\n");
        System.out.print("Querying index " + indexName + "...");

        Index index = table.getIndex(indexName);

        ItemCollection<QueryOutcome> items = null;

        QuerySpec querySpec = new QuerySpec();

        if (indexName == "CreateDateIndex") {
            System.out.println("Issues filed on 2013-11-01");
            querySpec.withKeyConditionExpression("CreateDate = :v_date and
begins_with(IssueId, :v_issue)")
                    .withValueMap(new ValueMap().withString(":v_date",
"2013-11-01").withString(":v_issue", "A-"));
            items = index.query(querySpec);
        } else if (indexName == "TitleIndex") {
            System.out.println("Compilation errors");
            querySpec.withKeyConditionExpression("Title = :v_title and
begins_with(IssueId, :v_issue)")
                    .withValueMap(
                            new ValueMap().withString(":v_title", "Compilation
error").withString(":v_issue", "A-"));
            items = index.query(querySpec);
        } else if (indexName == "DueDateIndex") {
            System.out.println("Items that are due on 2013-11-30");
            querySpec.withKeyConditionExpression("DueDate = :v_date")
                    .withValueMap(new ValueMap().withString(":v_date", "2013-11-30"));
            items = index.query(querySpec);
        } else {
            System.out.println("\nNo valid index name provided");
            return;
        }

        Iterator<Item> iterator = items.iterator();
```

```
        System.out.println("Query: printing results...");

        while (iterator.hasNext()) {
            System.out.println(iterator.next().toJSONPretty());
        }

    }

    public static void deleteTable(String tableName) {

        System.out.println("Deleting table " + tableName + "...");

        Table table = dynamoDB.getTable(tableName);
        table.delete();

        // Wait for table to be deleted
        System.out.println("Waiting for " + tableName + " to be deleted...");
        try {
            table.waitForDelete();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    public static void loadData() {

        System.out.println("Loading data into table " + tableName + "...");

        // IssueId, Title,
        // Description,
        // CreateDate, LastUpdateDate, DueDate,
        // Priority, Status

        putItem("A-101", "Compilation error", "Can't compile Project X - bad version
number. What does this mean?",
                "2013-11-01", "2013-11-02", "2013-11-10", 1, "Assigned");

        putItem("A-102", "Can't read data file", "The main data file is missing, or the
permissions are incorrect",
                "2013-11-01", "2013-11-04", "2013-11-30", 2, "In progress");

        putItem("A-103", "Test failure", "Functional test of Project X produces
errors", "2013-11-01", "2013-11-02",
                "2013-11-10", 1, "In progress");
```

```
        putItem("A-104", "Compilation error", "Variable 'messageCount' was not
 initialized.", "2013-11-15",
                "2013-11-16", "2013-11-30", 3, "Assigned");

        putItem("A-105", "Network issue", "Can't ping IP address 127.0.0.1. Please fix
 this.", "2013-11-15",
                "2013-11-16", "2013-11-19", 5, "Assigned");

    }

    public static void putItem(

            String issueId, String title, String description, String createDate, String
 lastUpdateDate, String dueDate,
            Integer priority, String status) {

        Table table = dynamoDB.getTable(tableName);

        Item item = new Item().withPrimaryKey("IssueId", issueId).withString("Title",
 title)
                .withString("Description", description).withString("CreateDate",
 createDate)
                .withString("LastUpdateDate", lastUpdateDate).withString("DueDate",
 dueDate)
                .withNumber("Priority", priority).withString("Status", status);

        table.putItem(item);
    }

}
```

Working with Global Secondary Indexes: .NET

You can use the AWS SDK for .NET low-level API to create an Amazon DynamoDB table with one or more global secondary indexes, describe the indexes on the table, and perform queries using the indexes. These operations map to the corresponding DynamoDB operations. For more information, see the Amazon DynamoDB API Reference.

The following are the common steps for table operations using the .NET low-level API.

1. Create an instance of the `AmazonDynamoDBClient` class.

2. Provide the required and optional parameters for the operation by creating the corresponding request objects.

   For example, create a `CreateTableRequest` object to create a table and `QueryRequest` object to query a table or an index.
3. Run the appropriate method provided by the client that you created in the preceding step.

Topics

- Create a table with a Global Secondary Index
- Describe a table with a Global Secondary Index
- Query a Global Secondary Index
- Example: Global Secondary Indexes using the AWS SDK for .NET low-level API

Create a table with a Global Secondary Index

You can create global secondary indexes at the same time that you create a table. To do this, use `CreateTable` and provide your specifications for one or more global secondary indexes. The following C# code example creates a table to hold information about weather data. The partition key is `Location` and the sort key is `Date`. A global secondary index named `PrecipIndex` allows fast access to precipitation data for various locations.

The following are the steps to create a table with a global secondary index, using the .NET low-level API.

1. Create an instance of the `AmazonDynamoDBClient` class.
2. Create an instance of the `CreateTableRequest` class to provide the request information.

   You must provide the table name, its primary key, and the provisioned throughput values. For the global secondary index, you must provide the index name, its provisioned throughput settings, the attribute definitions for the index sort key, the key schema for the index, and the attribute projection.
3. Run the `CreateTable` method by providing the request object as a parameter.

The following C# code example demonstrates the preceding steps. The code creates a table (`WeatherData`) with a global secondary index (`PrecipIndex`). The index partition key is `Date` and its sort key is `Precipitation`. All of the table attributes are projected into the index. Users

can query this index to obtain weather data for a particular date, optionally sorting the data by precipitation amount.

Because `Precipitation` is not a key attribute for the table, it is not required. However, `WeatherData` items without `Precipitation` do not appear in `PrecipIndex`.

```
client = new AmazonDynamoDBClient();
string tableName = "WeatherData";

// Attribute definitions
var attributeDefinitions = new List<AttributeDefinition>()
{
    {new AttributeDefinition{
        AttributeName = "Location",
        AttributeType = "S"}},
    {new AttributeDefinition{
        AttributeName = "Date",
        AttributeType = "S"}},
    {new AttributeDefinition(){
        AttributeName = "Precipitation",
        AttributeType = "N"}
    }
};

// Table key schema
var tableKeySchema = new List<KeySchemaElement>()
{
    {new KeySchemaElement {
        AttributeName = "Location",
        KeyType = "HASH"}},  //Partition key
    {new KeySchemaElement {
        AttributeName = "Date",
        KeyType = "RANGE"}  //Sort key
    }
};

// PrecipIndex
var precipIndex = new GlobalSecondaryIndex
{
    IndexName = "PrecipIndex",
    ProvisionedThroughput = new ProvisionedThroughput
    {
        ReadCapacityUnits = (long)10,
```

```
        WriteCapacityUnits = (long)1
    },
    Projection = new Projection { ProjectionType = "ALL" }
};

var indexKeySchema = new List<KeySchemaElement> {
    {new KeySchemaElement { AttributeName = "Date", KeyType = "HASH"}},  //Partition
 key
    {new KeySchemaElement{AttributeName = "Precipitation",KeyType = "RANGE"}}  //Sort
 key
};

precipIndex.KeySchema = indexKeySchema;

CreateTableRequest createTableRequest = new CreateTableRequest
{
    TableName = tableName,
    ProvisionedThroughput = new ProvisionedThroughput
    {
        ReadCapacityUnits = (long)5,
        WriteCapacityUnits = (long)1
    },
    AttributeDefinitions = attributeDefinitions,
    KeySchema = tableKeySchema,
    GlobalSecondaryIndexes = { precipIndex }
};

CreateTableResponse response = client.CreateTable(createTableRequest);
Console.WriteLine(response.CreateTableResult.TableDescription.TableName);
Console.WriteLine(response.CreateTableResult.TableDescription.TableStatus);
```

You must wait until DynamoDB creates the table and sets the table status to `ACTIVE`. After that, you can begin putting data items into the table.

Describe a table with a Global Secondary Index

To get information about global secondary indexes on a table, use `DescribeTable`. For each index, you can access its name, key schema, and projected attributes.

The following are the steps to access global secondary index information for a table using the .NET low-level API.

1. Create an instance of the `AmazonDynamoDBClient` class.

2. Run the `describeTable` method by providing the request object as a parameter.

   Create an instance of the `DescribeTableRequest` class to provide the request information. You must provide the table name.

3.

The following C# code example demonstrates the preceding steps.

Example

```
client = new AmazonDynamoDBClient();
string tableName = "WeatherData";

DescribeTableResponse response = client.DescribeTable(new DescribeTableRequest
 { TableName = tableName});

List<GlobalSecondaryIndexDescription> globalSecondaryIndexes =
response.DescribeTableResult.Table.GlobalSecondaryIndexes;

// This code snippet will work for multiple indexes, even though
// there is only one index in this example.

foreach (GlobalSecondaryIndexDescription gsiDescription in globalSecondaryIndexes) {
     Console.WriteLine("Info for index " + gsiDescription.IndexName + ":");

     foreach (KeySchemaElement kse in gsiDescription.KeySchema) {
          Console.WriteLine("\t" + kse.AttributeName + ": key type is " + kse.KeyType);
     }

      Projection projection = gsiDescription.Projection;
      Console.WriteLine("\tThe projection type is: " + projection.ProjectionType);

      if (projection.ProjectionType.ToString().Equals("INCLUDE")) {
           Console.WriteLine("\t\tThe non-key projected attributes are: "
                + projection.NonKeyAttributes);
      }
}
```

Query a Global Secondary Index

You can use `Query` on a global secondary index, in much the same way you `Query` a table. You need to specify the index name, the query criteria for the index partition key and sort key (if

present), and the attributes that you want to return. In this example, the index is `PrecipIndex`, which has a partition key of `Date` and a sort key of `Precipitation`. The index query returns all of the weather data for a particular date, where the precipitation is greater than zero.

The following are the steps to query a global secondary index using the .NET low-level API.

1. Create an instance of the `AmazonDynamoDBClient` class.
2. Create an instance of the `QueryRequest` class to provide the request information.
3. Run the `query` method by providing the request object as a parameter.

The attribute name `Date` is a DynamoDB reserved word. Therefore, you must use an expression attribute name as a placeholder in the `KeyConditionExpression`.

The following C# code example demonstrates the preceding steps.

Example

```
client = new AmazonDynamoDBClient();

QueryRequest queryRequest = new QueryRequest
{
    TableName = "WeatherData",
    IndexName = "PrecipIndex",
    KeyConditionExpression = "#dt = :v_date and Precipitation > :v_precip",
    ExpressionAttributeNames = new Dictionary<String, String> {
        {"#dt", "Date"}
    },
    ExpressionAttributeValues = new Dictionary<string, AttributeValue> {
        {":v_date", new AttributeValue { S =  "2013-08-01" }},
        {":v_precip", new AttributeValue { N =  "0" }}
    },
    ScanIndexForward = true
};

var result = client.Query(queryRequest);

var items = result.Items;
foreach (var currentItem in items)
{
    foreach (string attr in currentItem.Keys)
    {
        Console.Write(attr + "---> ");
```

```
        if (attr == "Precipitation")
        {
            Console.WriteLine(currentItem[attr].N);
    }
    else
    {
        Console.WriteLine(currentItem[attr].S);
    }

         }
     Console.WriteLine();
}
```

Example: Global Secondary Indexes using the AWS SDK for .NET low-level API

The following C# code example shows how to work with global secondary indexes. The example creates a table named `Issues`, which might be used in a simple bug tracking system for software development. The partition key is `IssueId` and the sort key is `Title`. There are three global secondary indexes on this table:

- `CreateDateIndex` — The partition key is `CreateDate` and the sort key is `IssueId`. In addition to the table keys, the attributes `Description` and `Status` are projected into the index.
- `TitleIndex` — The partition key is `Title` and the sort key is `IssueId`. No attributes other than the table keys are projected into the index.
- `DueDateIndex` — The partition key is `DueDate`, and there is no sort key. All of the table attributes are projected into the index.

After the `Issues` table is created, the program loads the table with data representing software bug reports. It then queries the data using the global secondary indexes. Finally, the program deletes the `Issues` table.

For step-by-step instructions for testing the following sample, see .NET code examples.

Example

```
using System;
using System.Collections.Generic;
using System.Linq;
using Amazon.DynamoDBv2;
using Amazon.DynamoDBv2.DataModel;
```

```
using Amazon.DynamoDBv2.DocumentModel;
using Amazon.DynamoDBv2.Model;
using Amazon.Runtime;
using Amazon.SecurityToken;

namespace com.amazonaws.codesamples
{
    class LowLevelGlobalSecondaryIndexExample
    {
        private static AmazonDynamoDBClient client = new AmazonDynamoDBClient();
        public static String tableName = "Issues";

        public static void Main(string[] args)
        {
            CreateTable();
            LoadData();

            QueryIndex("CreateDateIndex");
            QueryIndex("TitleIndex");
            QueryIndex("DueDateIndex");

            DeleteTable(tableName);

            Console.WriteLine("To continue, press enter");
            Console.Read();
        }

        private static void CreateTable()
        {
            // Attribute definitions
            var attributeDefinitions = new List<AttributeDefinition>()
        {
            {new AttributeDefinition {
                 AttributeName = "IssueId", AttributeType = "S"
             }},
            {new AttributeDefinition {
                 AttributeName = "Title", AttributeType = "S"
             }},
            {new AttributeDefinition {
                 AttributeName = "CreateDate", AttributeType = "S"
             }},
            {new AttributeDefinition {
                 AttributeName = "DueDate", AttributeType = "S"
             }}
```

```
        };

            // Key schema for table
            var tableKeySchema = new List<KeySchemaElement>() {
            {
                new KeySchemaElement {
                    AttributeName= "IssueId",
                    KeyType = "HASH" //Partition key
                }
            },
            {
                new KeySchemaElement {
                    AttributeName = "Title",
                    KeyType = "RANGE" //Sort key
                }
            }
        };

            // Initial provisioned throughput settings for the indexes
            var ptIndex = new ProvisionedThroughput
            {
                ReadCapacityUnits = 1L,
                WriteCapacityUnits = 1L
            };

            // CreateDateIndex
            var createDateIndex = new GlobalSecondaryIndex()
            {
                IndexName = "CreateDateIndex",
                ProvisionedThroughput = ptIndex,
                KeySchema = {
                new KeySchemaElement {
                    AttributeName = "CreateDate", KeyType = "HASH" //Partition key
                },
                new KeySchemaElement {
                    AttributeName = "IssueId", KeyType = "RANGE" //Sort key
                }
            },
                Projection = new Projection
                {
                    ProjectionType = "INCLUDE",
                    NonKeyAttributes = {
                    "Description", "Status"
                }
```

```
            }
        };

        // TitleIndex
        var titleIndex = new GlobalSecondaryIndex()
        {
            IndexName = "TitleIndex",
            ProvisionedThroughput = ptIndex,
            KeySchema = {
            new KeySchemaElement {
                AttributeName = "Title", KeyType = "HASH" //Partition key
            },
            new KeySchemaElement {
                AttributeName = "IssueId", KeyType = "RANGE" //Sort key
            }
        },
            Projection = new Projection
            {
                ProjectionType = "KEYS_ONLY"
            }
        };

        // DueDateIndex
        var dueDateIndex = new GlobalSecondaryIndex()
        {
            IndexName = "DueDateIndex",
            ProvisionedThroughput = ptIndex,
            KeySchema = {
            new KeySchemaElement {
                AttributeName = "DueDate",
                KeyType = "HASH" //Partition key
            }
        },
            Projection = new Projection
            {
                ProjectionType = "ALL"
            }
        };



        var createTableRequest = new CreateTableRequest
        {
            TableName = tableName,
```

```
            ProvisionedThroughput = new ProvisionedThroughput
            {
                ReadCapacityUnits = (long)1,
                WriteCapacityUnits = (long)1
            },
            AttributeDefinitions = attributeDefinitions,
            KeySchema = tableKeySchema,
            GlobalSecondaryIndexes = {
            createDateIndex, titleIndex, dueDateIndex
        }
        };

        Console.WriteLine("Creating table " + tableName + "...");
        client.CreateTable(createTableRequest);

        WaitUntilTableReady(tableName);
    }

    private static void LoadData()
    {
        Console.WriteLine("Loading data into table " + tableName + "...");

        // IssueId, Title,
        // Description,
        // CreateDate, LastUpdateDate, DueDate,
        // Priority, Status

        putItem("A-101", "Compilation error",
            "Can't compile Project X - bad version number. What does this mean?",
            "2013-11-01", "2013-11-02", "2013-11-10",
            1, "Assigned");

        putItem("A-102", "Can't read data file",
            "The main data file is missing, or the permissions are incorrect",
            "2013-11-01", "2013-11-04", "2013-11-30",
            2, "In progress");

        putItem("A-103", "Test failure",
            "Functional test of Project X produces errors",
            "2013-11-01", "2013-11-02", "2013-11-10",
            1, "In progress");

        putItem("A-104", "Compilation error",
            "Variable 'messageCount' was not initialized.",
```

```
                "2013-11-15", "2013-11-16", "2013-11-30",
                3, "Assigned");

            putItem("A-105", "Network issue",
                "Can't ping IP address 127.0.0.1. Please fix this.",
                "2013-11-15", "2013-11-16", "2013-11-19",
                5, "Assigned");
        }

        private static void putItem(
            String issueId, String title,
            String description,
            String createDate, String lastUpdateDate, String dueDate,
            Int32 priority, String status)
        {
            Dictionary<String, AttributeValue> item = new Dictionary<string,
AttributeValue>();

            item.Add("IssueId", new AttributeValue
            {
                S = issueId
            });
            item.Add("Title", new AttributeValue
            {
                S = title
            });
            item.Add("Description", new AttributeValue
            {
                S = description
            });
            item.Add("CreateDate", new AttributeValue
            {
                S = createDate
            });
            item.Add("LastUpdateDate", new AttributeValue
            {
                S = lastUpdateDate
            });
            item.Add("DueDate", new AttributeValue
            {
                S = dueDate
            });
            item.Add("Priority", new AttributeValue
            {
```

```
                N = priority.ToString()
            });
            item.Add("Status", new AttributeValue
            {
                S = status
            });

            try
            {
                client.PutItem(new PutItemRequest
                {
                    TableName = tableName,
                    Item = item
                });
            }
            catch (Exception e)
            {
                Console.WriteLine(e.ToString());
            }
        }

        private static void QueryIndex(string indexName)
        {
            Console.WriteLine
                ("\n*******************************************************\n");
            Console.WriteLine("Querying index " + indexName + "...");

            QueryRequest queryRequest = new QueryRequest
            {
                TableName = tableName,
                IndexName = indexName,
                ScanIndexForward = true
            };


            String keyConditionExpression;
            Dictionary<string, AttributeValue> expressionAttributeValues = new
 Dictionary<string, AttributeValue>();

            if (indexName == "CreateDateIndex")
            {
                Console.WriteLine("Issues filed on 2013-11-01\n");
```

```
                keyConditionExpression = "CreateDate = :v_date and
begins_with(IssueId, :v_issue)";
                expressionAttributeValues.Add(":v_date", new AttributeValue
                {
                    S = "2013-11-01"
                });
                expressionAttributeValues.Add(":v_issue", new AttributeValue
                {
                    S = "A-"
                });
            }
            else if (indexName == "TitleIndex")
            {
                Console.WriteLine("Compilation errors\n");

                keyConditionExpression = "Title = :v_title and
begins_with(IssueId, :v_issue)";
                expressionAttributeValues.Add(":v_title", new AttributeValue
                {
                    S = "Compilation error"
                });
                expressionAttributeValues.Add(":v_issue", new AttributeValue
                {
                    S = "A-"
                });

                // Select
                queryRequest.Select = "ALL_PROJECTED_ATTRIBUTES";
            }
            else if (indexName == "DueDateIndex")
            {
                Console.WriteLine("Items that are due on 2013-11-30\n");

                keyConditionExpression = "DueDate = :v_date";
                expressionAttributeValues.Add(":v_date", new AttributeValue
                {
                    S = "2013-11-30"
                });

                // Select
                queryRequest.Select = "ALL_PROJECTED_ATTRIBUTES";
            }
            else
            {
```

```
            Console.WriteLine("\nNo valid index name provided");
            return;
        }

        queryRequest.KeyConditionExpression = keyConditionExpression;
        queryRequest.ExpressionAttributeValues = expressionAttributeValues;

        var result = client.Query(queryRequest);
        var items = result.Items;
        foreach (var currentItem in items)
        {
            foreach (string attr in currentItem.Keys)
            {
                if (attr == "Priority")
                {
                    Console.WriteLine(attr + "---> " + currentItem[attr].N);
                }
                else
                {
                    Console.WriteLine(attr + "---> " + currentItem[attr].S);
                }
            }
            Console.WriteLine();
        }
    }

    private static void DeleteTable(string tableName)
    {
        Console.WriteLine("Deleting table " + tableName + "...");
        client.DeleteTable(new DeleteTableRequest
        {
            TableName = tableName
        });
        WaitForTableToBeDeleted(tableName);
    }

    private static void WaitUntilTableReady(string tableName)
    {
        string status = null;
        // Let us wait until table is created. Call DescribeTable.
        do
        {
            System.Threading.Thread.Sleep(5000); // Wait 5 seconds.
            try
```

```
            {
                var res = client.DescribeTable(new DescribeTableRequest
                {
                    TableName = tableName
                });

                Console.WriteLine("Table name: {0}, status: {1}",
                          res.Table.TableName,
                          res.Table.TableStatus);
                status = res.Table.TableStatus;
            }
            catch (ResourceNotFoundException)
            {
                // DescribeTable is eventually consistent. So you might
                // get resource not found. So we handle the potential exception.
            }
        } while (status != "ACTIVE");
    }

    private static void WaitForTableToBeDeleted(string tableName)
    {
        bool tablePresent = true;

        while (tablePresent)
        {
            System.Threading.Thread.Sleep(5000); // Wait 5 seconds.
            try
            {
                var res = client.DescribeTable(new DescribeTableRequest
                {
                    TableName = tableName
                });

                Console.WriteLine("Table name: {0}, status: {1}",
                          res.Table.TableName,
                          res.Table.TableStatus);
            }
            catch (ResourceNotFoundException)
            {
                tablePresent = false;
            }
        }
    }
}
```

```
}
```

Working with Global Secondary Indexes in DynamoDB using AWS CLI

You can use the AWS CLI to create an Amazon DynamoDB table with one or more global secondary indexes, describe the indexes on the table, and perform queries using the indexes.

Topics

- Create a table with a Global Secondary Index
- Add a Global Secondary Index to an existing table
- Describe a table with a Global Secondary Index
- Query a Global Secondary Index

Create a table with a Global Secondary Index

Global secondary indexes may be created at the same time you create a table. To do this, use the `create-table` parameter and provide your specifications for one or more global secondary indexes. The following example creates a table named `GameScores` with a global secondary index called `GameTitleIndex`. The base table has a partition key of `UserId` and a sort key of `GameTitle`, allowing you to find an individual user's best score for a specific game efficiently, whereas the GSI has a partition key of `GameTitle` and a sort key of `TopScore`, allowing you to quickly find the overall highest score for a particular game.

```
aws dynamodb create-table \
    --table-name GameScores \
    --attribute-definitions AttributeName=UserId,AttributeType=S \
                            AttributeName=GameTitle,AttributeType=S \
                            AttributeName=TopScore,AttributeType=N  \
    --key-schema AttributeName=UserId,KeyType=HASH \
                 AttributeName=GameTitle,KeyType=RANGE \
    --provisioned-throughput ReadCapacityUnits=10,WriteCapacityUnits=5 \
    --global-secondary-indexes \
        "[
            {
                \"IndexName\": \"GameTitleIndex\",
                \"KeySchema\": [{\"AttributeName\":\"GameTitle\",\"KeyType\":\"HASH\"},
                                {\"AttributeName\":\"TopScore\",\"KeyType\":\"RANGE
\"}],
```

```
                \"Projection\":{
                    \"ProjectionType\":\"INCLUDE\",
                    \"NonKeyAttributes\":[\"UserId\"]
                },
                \"ProvisionedThroughput\": {
                    \"ReadCapacityUnits\": 10,
                    \"WriteCapacityUnits\": 5
                }
            }
        ]"
```

You must wait until DynamoDB creates the table and sets the table status to `ACTIVE`. After that, you can begin putting data items into the table. You can use describe-table to determine the status of the table creation.

Add a Global Secondary Index to an existing table

Global secondary indexes may also be added or modified after table creation. To do this, use the `update-table` parameter and provide your specifications for one or more global secondary indexes. The following example uses the same schema as the previous example, but assumes that the table has already been created and we're adding the GSI later.

```
aws dynamodb update-table \
    --table-name GameScores \
    --attribute-definitions AttributeName=TopScore,AttributeType=N  \
    --global-secondary-index-updates \
        "[
            {
                \"Create\": {
                    \"IndexName\": \"GameTitleIndex\",
                    \"KeySchema\": [{\"AttributeName\":\"GameTitle\",\"KeyType\":\"HASH
\"},
                                   {\"AttributeName\":\"TopScore\",\"KeyType\":\"RANGE
\"}],
                    \"Projection\":{
                        \"ProjectionType\":\"INCLUDE\",
                        \"NonKeyAttributes\":[\"UserId\"]
                    }
                }
            }
        ]"
```

Describe a table with a Global Secondary Index

To get information about Global Secondary Indexes on a table, use the `describe-table` parameter. For each index, you can access its name, key schema, and projected attributes.

```
aws dynamodb describe-table --table-name GameScores
```

Query a Global Secondary Index

You can use the `query` operation on a global secondary index in much the same way that you `query` a table. You must specify the index name, the query criteria for the index sort key, and the attributes that you want to return. In this example, the index is `GameTitleIndex` and the index sort key is `GameTitle`.

The only attributes returned are those that have been projected into the index. You could modify this query to select non-key attributes too, but this would require table fetch activity that is relatively expensive. For more information about table fetches, see Attribute projections.

```
aws dynamodb query --table-name GameScores\
    --index-name GameTitleIndex \
    --key-condition-expression "GameTitle = :v_game" \
    --expression-attribute-values '{":v_game":{"S":"Alien Adventure"} }'
```

Local secondary indexes

Some applications only need to query data using the base table's primary key. However, there might be situations where an alternative sort key would be helpful. To give your application a choice of sort keys, you can create one or more local secondary indexes on an Amazon DynamoDB table and issue `Query` or `Scan` requests against these indexes.

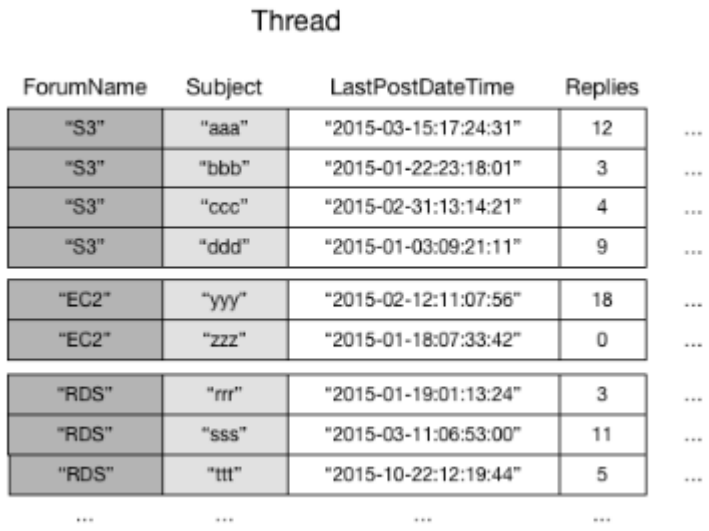
Topics

- Scenario: Using a Local Secondary Index
- Attribute projections
- Creating a Local Secondary Index
- Reading data from a Local Secondary Index
- Item writes and Local Secondary Indexes
- Provisioned throughput considerations for Local Secondary Indexes
- Storage considerations for Local Secondary Indexes

- Item collections in Local Secondary Indexes
- Working with Local Secondary Indexes: Java
- Working with Local Secondary Indexes: .NET
- Working with Local Secondary Indexes in DynamoDB AWS CLI

Scenario: Using a Local Secondary Index

As an example, consider the `Thread` table. This table is useful for an application such as the AWS discussion forums. The following diagram shows how the items in the table would be organized. (Not all of the attributes are shown.)



DynamoDB stores all of the items with the same partition key value continuously. In this example, given a particular `ForumName`, a `Query` operation could immediately locate all of the threads for that forum. Within a group of items with the same partition key value, the items are sorted by sort key value. If the sort key (`Subject`) is also provided in the query, DynamoDB can narrow down the results that are returned—for example, returning all of the threads in the "S3" forum that have a `Subject` beginning with the letter "a".

Some requests might require more complex data access patterns. For example:

- Which forum threads get the most views and replies?
- Which thread in a particular forum has the largest number of messages?
- How many threads were posted in a particular forum within a particular time period?

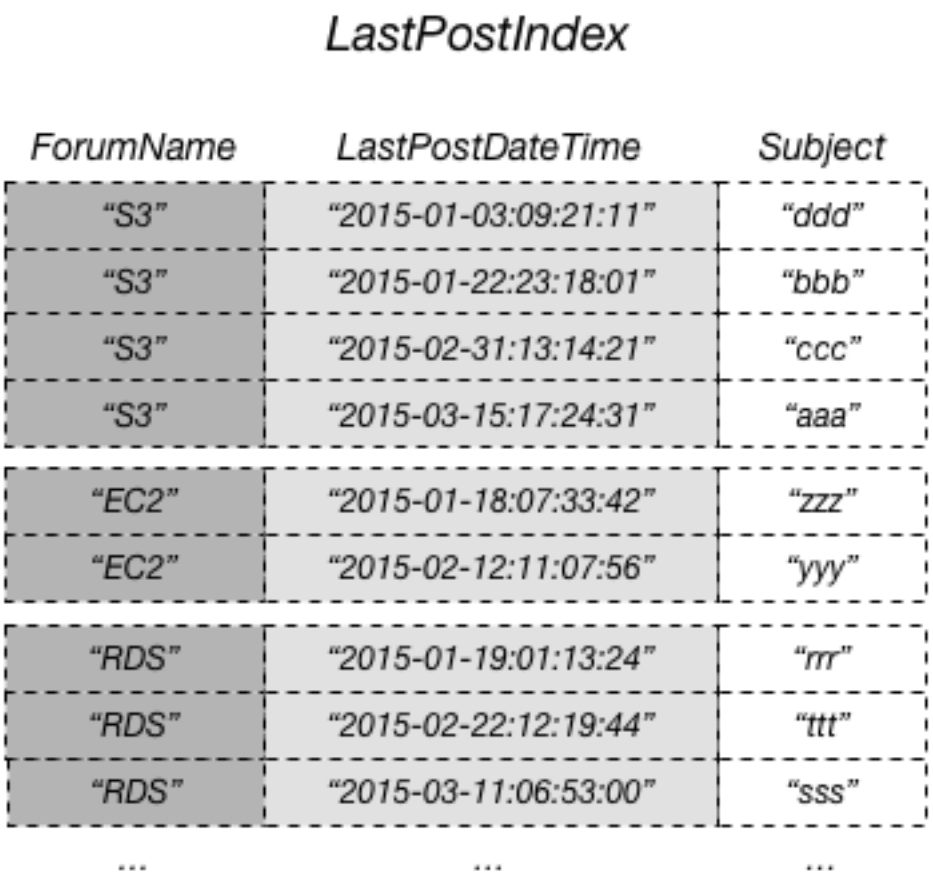
To answer these questions, the `Query` action would not be sufficient. Instead, you would have to `Scan` the entire table. For a table with millions of items, this would consume a large amount of provisioned read throughput and take a long time to complete.

However, you can specify one or more local secondary indexes on non-key attributes, such as `Replies` or `LastPostDateTime`.

A *local secondary index* maintains an alternate sort key for a given partition key value. A local secondary index also contains a copy of some or all of the attributes from its base table. You specify which attributes are projected into the local secondary index when you create the table. The data in a local secondary index is organized by the same partition key as the base table, but with a different sort key. This lets you access data items efficiently across this different dimension. For greater query or scan flexibility, you can create up to five local secondary indexes per table.

Suppose that an application needs to find all of the threads that have been posted within the last three months in a particular forum. Without a local secondary index, the application would have to `Scan` the entire `Thread` table and discard any posts that were not within the specified time frame. With a local secondary index, a `Query` operation could use `LastPostDateTime` as a sort key and find the data quickly.

The following diagram shows a local secondary index named `LastPostIndex`. Note that the partition key is the same as that of the `Thread` table, but the sort key is `LastPostDateTime`.



Every local secondary index must meet the following conditions:

- The partition key is the same as that of its base table.
- The sort key consists of exactly one scalar attribute.
- The sort key of the base table is projected into the index, where it acts as a non-key attribute.

In this example, the partition key is `ForumName` and the sort key of the local secondary index is `LastPostDateTime`. In addition, the sort key value from the base table (in this example, `Subject`) is projected into the index, but it is not a part of the index key. If an application needs a list that is based on `ForumName` and `LastPostDateTime`, it can issue a `Query` request against `LastPostIndex`. The query results are sorted by `LastPostDateTime`, and can be returned in ascending or descending order. The query can also apply key conditions, such as returning only items that have a `LastPostDateTime` within a particular time span.

Every local secondary index automatically contains the partition and sort keys from its base table; you can optionally project non-key attributes into the index. When you query the index, DynamoDB can retrieve these projected attributes efficiently. When you query a local secondary index, the

query can also retrieve attributes that are *not* projected into the index. DynamoDB automatically fetches these attributes from the base table, but at a greater latency and with higher provisioned throughput costs.

For any local secondary index, you can store up to 10 GB of data per distinct partition key value. This figure includes all of the items in the base table, plus all of the items in the indexes, that have the same partition key value. For more information, see Item collections in Local Secondary Indexes.

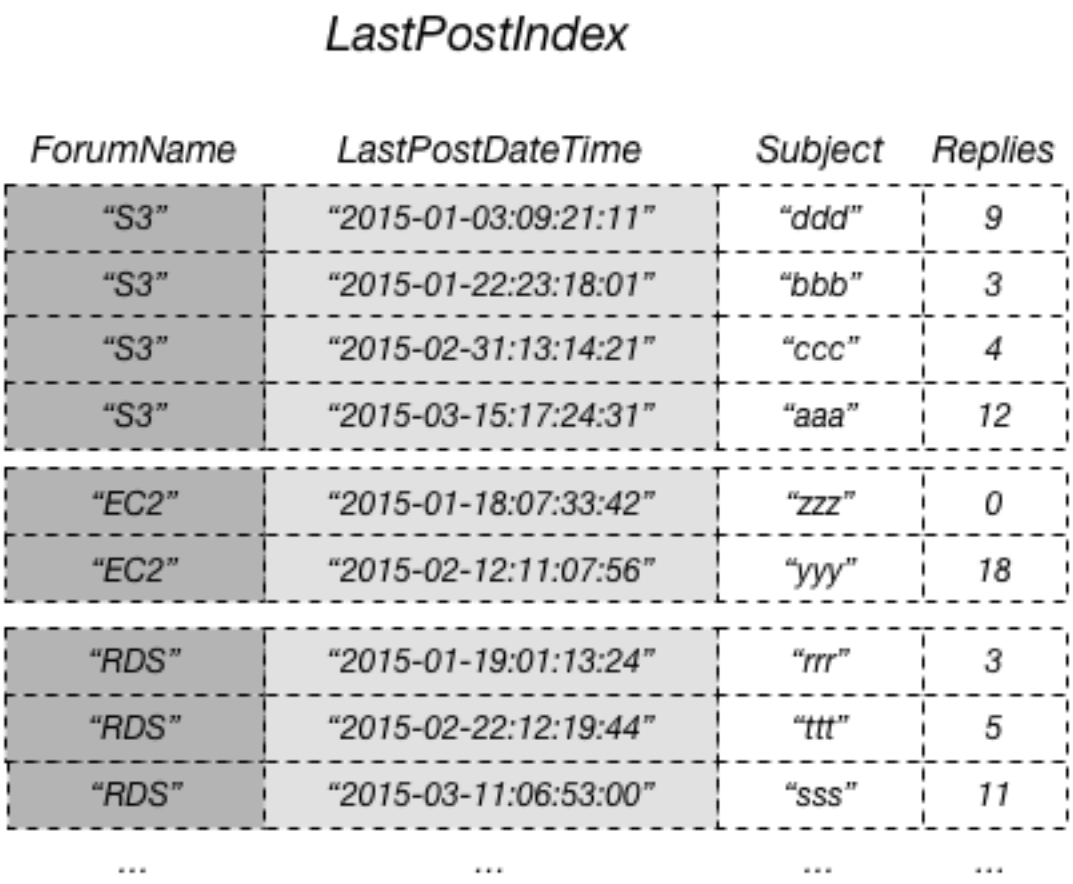
Attribute projections

With `LastPostIndex`, an application could use `ForumName` and `LastPostDateTime` as query criteria. However, to retrieve any additional attributes, DynamoDB must perform additional read operations against the `Thread` table. These extra reads are known as *fetches*, and they can increase the total amount of provisioned throughput required for a query.

Suppose that you wanted to populate a webpage with a list of all the threads in "S3" and the number of replies for each thread, sorted by the last reply date/time beginning with the most recent reply. To populate this list, you would need the following attributes:

- `Subject`
- `Replies`
- `LastPostDateTime`

The most efficient way to query this data and to avoid fetch operations would be to project the `Replies` attribute from the table into the local secondary index, as shown in this diagram.



A *projection* is the set of attributes that is copied from a table into a secondary index. The partition key and sort key of the table are always projected into the index; you can project other attributes to support your application's query requirements. When you query an index, Amazon DynamoDB can access any attribute in the projection as if those attributes were in a table of their own.

When you create a secondary index, you need to specify the attributes that will be projected into the index. DynamoDB provides three different options for this:

- *KEYS_ONLY* – Each item in the index consists only of the table partition key and sort key values, plus the index key values. The `KEYS_ONLY` option results in the smallest possible secondary index.
- *INCLUDE* – In addition to the attributes described in `KEYS_ONLY`, the secondary index will include other non-key attributes that you specify.
- *ALL* – The secondary index includes all of the attributes from the source table. Because all of the table data is duplicated in the index, an `ALL` projection results in the largest possible secondary index.

In the previous diagram, the non-key attribute `Replies` is projected into `LastPostIndex`. An application can query `LastPostIndex` instead of the full `Thread` table to populate a webpage with `Subject`, `Replies`, and `LastPostDateTime`. If any other non-key attributes are requested, DynamoDB would need to fetch those attributes from the `Thread` table.

From an application's point of view, fetching additional attributes from the base table is automatic and transparent, so there is no need to rewrite any application logic. However, such fetching can greatly reduce the performance advantage of using a local secondary index.

When you choose the attributes to project into a local secondary index, you must consider the tradeoff between provisioned throughput costs and storage costs:

- If you need to access just a few attributes with the lowest possible latency, consider projecting only those attributes into a local secondary index. The smaller the index, the less that it costs to store it, and the less your write costs are. If there are attributes that you occasionally need to fetch, the cost for provisioned throughput may well outweigh the longer-term cost of storing those attributes.
- If your application frequently accesses some non-key attributes, you should consider projecting those attributes into a local secondary index. The additional storage costs for the local secondary index offset the cost of performing frequent table scans.
- If you need to access most of the non-key attributes on a frequent basis, you can project these attributes—or even the entire base table— into a local secondary index. This gives you maximum flexibility and lowest provisioned throughput consumption, because no fetching would be required. However, your storage cost would increase, or even double if you are projecting all attributes.
- If your application needs to query a table infrequently, but must perform many writes or updates against the data in the table, consider projecting *KEYS_ONLY*. The local secondary index would be of minimal size, but would still be available when needed for query activity.

Creating a Local Secondary Index

To create one or more local secondary indexes on a table, use the `LocalSecondaryIndexes` parameter of the `CreateTable` operation. Local secondary indexes on a table are created when the table is created. When you delete a table, any local secondary indexes on that table are also deleted.

You must specify one non-key attribute to act as the sort key of the local secondary index. The attribute that you choose must be a scalar `String`, `Number`, or `Binary`. Other scalar types, document types, and set types are not allowed. For a complete list of data types, see Data types.

> **Important**
>
> For tables with local secondary indexes, there is a 10 GB size limit per partition key value. A table with local secondary indexes can store any number of items, as long as the total size for any one partition key value does not exceed 10 GB. For more information, see Item collection size limit.

You can project attributes of any data type into a local secondary index. This includes scalars, documents, and sets. For a complete list of data types, see Data types.

Reading data from a Local Secondary Index

You can retrieve items from a local secondary index using the `Query` and `Scan` operations. The `GetItem` and `BatchGetItem` operations can't be used on a local secondary index.

Querying a Local Secondary Index

In a DynamoDB table, the combined partition key value and sort key value for each item must be unique. However, in a local secondary index, the sort key value does not need to be unique for a given partition key value. If there are multiple items in the local secondary index that have the same sort key value, a `Query` operation returns all of the items that have the same partition key value. In the response, the matching items are not returned in any particular order.

You can query a local secondary index using either eventually consistent or strongly consistent reads. To specify which type of consistency you want, use the `ConsistentRead` parameter of the `Query` operation. A strongly consistent read from a local secondary index always returns the latest updated values. If the query needs to fetch additional attributes from the base table, those attributes will be consistent with respect to the index.

Example

Consider the following data returned from a `Query` that requests data from the discussion threads in a particular forum.

```
{
    "TableName": "Thread",
```

```
    "IndexName": "LastPostIndex",
    "ConsistentRead": false,
    "ProjectionExpression": "Subject, LastPostDateTime, Replies, Tags",
    "KeyConditionExpression":
        "ForumName = :v_forum and LastPostDateTime between :v_start and :v_end",
    "ExpressionAttributeValues": {
        ":v_start": {"S": "2015-08-31T00:00:00.000Z"},
        ":v_end": {"S": "2015-11-31T00:00:00.000Z"},
        ":v_forum": {"S": "EC2"}
    }
}
```

In this query:

- DynamoDB accesses `LastPostIndex`, using the `ForumName` partition key to locate the index items for "EC2". All of the index items with this key are stored adjacent to each other for rapid retrieval.
- Within this forum, DynamoDB uses the index to look up the keys that match the specified `LastPostDateTime` condition.
- Because the `Replies` attribute is projected into the index, DynamoDB can retrieve this attribute without consuming any additional provisioned throughput.
- The `Tags` attribute is not projected into the index, so DynamoDB must access the `Thread` table and fetch this attribute.
- The results are returned, sorted by `LastPostDateTime`. The index entries are sorted by partition key value and then by sort key value, and `Query` returns them in the order they are stored. (You can use the `ScanIndexForward` parameter to return the results in descending order.)

Because the `Tags` attribute is not projected into the local secondary index, DynamoDB must consume additional read capacity units to fetch this attribute from the base table. If you need to run this query often, you should project `Tags` into `LastPostIndex` to avoid fetching from the base table. However, if you needed to access `Tags` only occasionally, the additional storage cost for projecting `Tags` into the index might not be worthwhile.

Scanning a Local Secondary Index

You can use `Scan` to retrieve all of the data from a local secondary index. You must provide the base table name and the index name in the request. With a `Scan`, DynamoDB reads all of

the data in the index and returns it to the application. You can also request that only some of the data be returned, and that the remaining data should be discarded. To do this, use the `FilterExpression` parameter of the `Scan` API. For more information, see Filter expressions for scan.

Item writes and Local Secondary Indexes

DynamoDB automatically keeps all local secondary indexes synchronized with their respective base tables. Applications never write directly to an index. However, it is important that you understand the implications of how DynamoDB maintains these indexes.

When you create a local secondary index, you specify an attribute to serve as the sort key for the index. You also specify a data type for that attribute. This means that whenever you write an item to the base table, if the item defines an index key attribute, its type must match the index key schema's data type. In the case of `LastPostIndex`, the `LastPostDateTime` sort key in the index is defined as a `String` data type. If you try to add an item to the `Thread` table and specify a different data type for `LastPostDateTime` (such as `Number`), DynamoDB returns a `ValidationException` because of the data type mismatch.

There is no requirement for a one-to-one relationship between the items in a base table and the items in a local secondary index. In fact, this behavior can be advantageous for many applications.

A table with many local secondary indexes incurs higher costs for write activity than tables with fewer indexes. For more information, see Provisioned throughput considerations for Local Secondary Indexes.

> **Important**
>
> For tables with local secondary indexes, there is a 10 GB size limit per partition key value. A table with local secondary indexes can store any number of items, as long as the total size for any one partition key value does not exceed 10 GB. For more information, see Item collection size limit.

Provisioned throughput considerations for Local Secondary Indexes

When you create a table in DynamoDB, you provision read and write capacity units for the table's expected workload. That workload includes read and write activity on the table's local secondary indexes.

To view the current rates for provisioned throughput capacity, see Amazon DynamoDB pricing.

Read capacity units

When you query a local secondary index, the number of read capacity units consumed depends on how the data is accessed.

As with table queries, an index query can use either eventually consistent or strongly consistent reads depending on the value of `ConsistentRead`. One strongly consistent read consumes one read capacity unit; an eventually consistent read consumes only half of that. Thus, by choosing eventually consistent reads, you can reduce your read capacity unit charges.

For index queries that request only index keys and projected attributes, DynamoDB calculates the provisioned read activity in the same way as it does for queries against tables. The only difference is that the calculation is based on the sizes of the index entries, rather than the size of the item in the base table. The number of read capacity units is the sum of all projected attribute sizes across all of the items returned; the result is then rounded up to the next 4 KB boundary. For more information about how DynamoDB calculates provisioned throughput usage, see DynamoDB provisioned capacity mode.

For index queries that read attributes that are not projected into the local secondary index, DynamoDB needs to fetch those attributes from the base table, in addition to reading the projected attributes from the index. These fetches occur when you include any non-projected attributes in the `Select` or `ProjectionExpression` parameters of the `Query` operation. Fetching causes additional latency in query responses, and it also incurs a higher provisioned throughput cost: In addition to the reads from the local secondary index described previously, you are charged for read capacity units for every base table item fetched. This charge is for reading each entire item from the table, not just the requested attributes.

The maximum size of the results returned by a `Query` operation is 1 MB. This includes the sizes of all the attribute names and values across all of the items returned. However, if a Query against a local secondary index causes DynamoDB to fetch item attributes from the base table, the maximum size of the data in the results might be lower. In this case, the result size is the sum of:

- The size of the matching items in the index, rounded up to the next 4 KB.
- The size of each matching item in the base table, with each item individually rounded up to the next 4 KB.

Using this formula, the maximum size of the results returned by a Query operation is still 1 MB.

For example, consider a table where the size of each item is 300 bytes. There is a local secondary index on that table, but only 200 bytes of each item is projected into the index. Now suppose that you `Query` this index, that the query requires table fetches for each item, and that the query returns 4 items. DynamoDB sums up the following:

- The size of the matching items in the index: 200 bytes × 4 items = 800 bytes; this is then rounded up to 4 KB.
- The size of each matching item in the base table: (300 bytes, rounded up to 4 KB) × 4 items = 16 KB.

The total size of the data in the result is therefore 20 KB.

Write capacity units

When an item in a table is added, updated, or deleted, updating the local secondary indexes consumes provisioned write capacity units for the table. The total provisioned throughput cost for a write is the sum of write capacity units consumed by writing to the table and those consumed by updating the local secondary indexes.

The cost of writing an item to a local secondary index depends on several factors:

- If you write a new item to the table that defines an indexed attribute, or you update an existing item to define a previously undefined indexed attribute, one write operation is required to put the item into the index.
- If an update to the table changes the value of an indexed key attribute (from A to B), two writes are required: one to delete the previous item from the index and another write to put the new item into the index.
- If an item was present in the index, but a write to the table caused the indexed attribute to be deleted, one write is required to delete the old item projection from the index.
- If an item is not present in the index before or after the item is updated, there is no additional write cost for the index.

All of these factors assume that the size of each item in the index is less than or equal to the 1 KB item size for calculating write capacity units. Larger index entries require additional write capacity units. You can minimize your write costs by considering which attributes your queries need to return and projecting only those attributes into the index.

Storage considerations for Local Secondary Indexes

When an application writes an item to a table, DynamoDB automatically copies the correct subset of attributes to any local secondary indexes in which those attributes should appear. Your AWS account is charged for storage of the item in the base table and also for storage of attributes in any local secondary indexes on that table.

The amount of space used by an index item is the sum of the following:

- The size in bytes of the base table primary key (partition key and sort key)
- The size in bytes of the index key attribute
- The size in bytes of the projected attributes (if any)
- 100 bytes of overhead per index item

To estimate the storage requirements for a local secondary index, you can estimate the average size of an item in the index and then multiply by the number of items in the index.

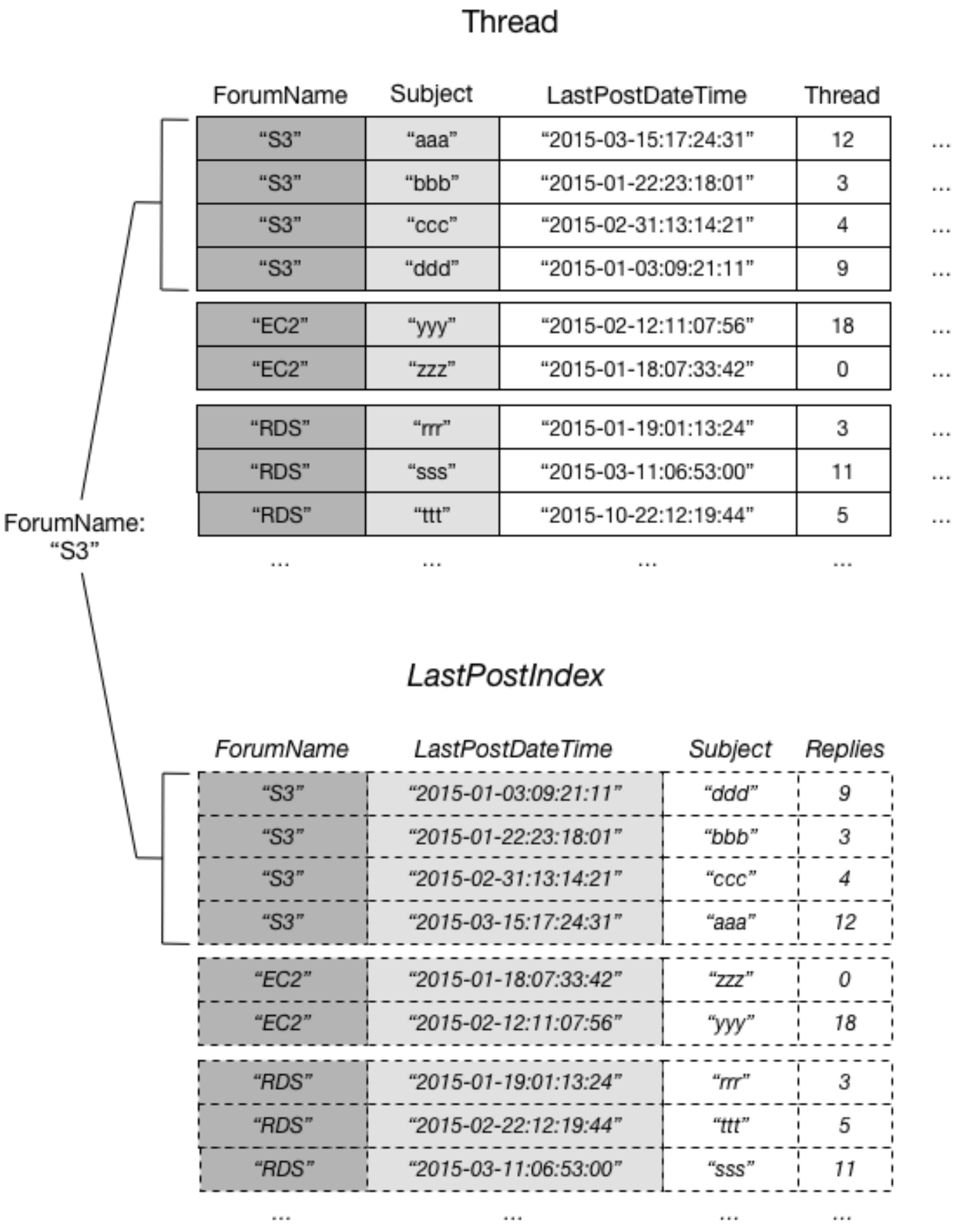
If a table contains an item where a particular attribute is not defined, but that attribute is defined as an index sort key, then DynamoDB does not write any data for that item to the index.

Item collections in Local Secondary Indexes

> **Note**
>
> This section pertains only to tables that have local secondary indexes.

In DynamoDB, an *item collection* is any group of items that have the same partition key value in a table and all of its local secondary indexes. In the examples used throughout this section, the partition key for the `Thread` table is `ForumName`, and the partition key for `LastPostIndex` is also `ForumName`. All the table and index items with the same `ForumName` are part of the same item collection. For example, in the `Thread` table and the `LastPostIndex` local secondary index, there is an item collection for forum `EC2` and a different item collection for forum `RDS`.

The following diagram shows the item collection for forum `S3`.


Thread
ForumName
Subject
LastPostDateTime
Thread
"S3"
"aaa"
"2015-03-15:17:24:31"
12
"S3"
"bbb"
"2015-01-22:23:18:01"
3
"S3"
"ccc"
"2015-02-31:13:14:21"
4
"S3"
"ddd"
"2015-01-03:09:21:11"
9
"EC2"
"yyy"
"2015-02-12:11:07:56"
18
"EC2"
"zzz"
"2015-01-18:07:33:42"
0
"RDS"
"rrr"
"2015-01-19:01:13:24"
3
"RDS"
"sss"
"2015-03-11:06:53:00"
11
"RDS"
"ttt"
"2015-10-22:12:19:44"
5
ForumName:
"S3"
LastPostIndex
ForumName
LastPostDateTime
Subject
Replies
"S3"
"2015-01-03:09:21:11"
"ddd"
9
"S3"
"2015-01-22:23:18:01"
"bbb"
3
"S3"
"2015-02-31:13:14:21"
"ccc"
4
"S3"
"2015-03-15:17:24:31"
"aaa"
12
"EC2"
"2015-01-18:07:33:42"
"zzz"
0
"EC2"
"2015-02-12:11:07:56"
"yyy"
18
"RDS"
"2015-01-19:01:13:24"
"rrr"
3
"RDS"
"2015-02-22:12:19:44"
"ttt"
5
"RDS"
"2015-03-11:06:53:00"
"sss"
11

In this diagram, the item collection consists of all the items in `Thread` and `LastPostIndex` where the `ForumName` partition key value is "S3". If there were other local secondary indexes on the table, any items in those indexes with `ForumName` equal to "S3" would also be part of the item collection.

You can use any of the following operations in DynamoDB to return information about item collections:

- `BatchWriteItem`
- `DeleteItem`
- `PutItem`
- `UpdateItem`
- `TransactWriteItems`

Each of these operations supports the `ReturnItemCollectionMetrics` parameter. When you set this parameter to `SIZE`, you can view information about the size of each item collection in the index.

Example

The following is an example from the output of an `UpdateItem` operation on the `Thread` table, with `ReturnItemCollectionMetrics` set to `SIZE`. The item that was updated had a `ForumName` value of "EC2", so the output includes information about that item collection.

```
{
    ItemCollectionMetrics: {
        ItemCollectionKey: {
            ForumName: "EC2"
        },
        SizeEstimateRangeGB: [0.0, 1.0]
    }
}
```

The `SizeEstimateRangeGB` object shows that the size of this item collection is between 0 and 1 GB. DynamoDB periodically updates this size estimate, so the numbers might be different next time the item is modified.

Item collection size limit

The maximum size of any item collection for a table which has one or more local secondary indexes is 10 GB. This does not apply to item collections in tables without local secondary indexes, and also does not apply to item collections in global secondary indexes. Only tables that have one or more local secondary indexes are affected.

If an item collection exceeds the 10 GB limit, DynamoDB may return an `ItemCollectionSizeLimitExceededException`, and you may not be able to add more items to the item collection or increase the sizes of items that are in the item collection. (Read and write operations that shrink the size of the item collection are still allowed.) You can still add items to other item collections.

To reduce the size of an item collection, you can do one of the following:

- Delete any unnecessary items with the partition key value in question. When you delete these items from the base table, DynamoDB also removes any index entries that have the same partition key value.
- Update the items by removing attributes or by reducing the size of the attributes. If these attributes are projected into any local secondary indexes, DynamoDB also reduces the size of the corresponding index entries.
- Create a new table with the same partition key and sort key, and then move items from the old table to the new table. This might be a good approach if a table has historical data that is infrequently accessed. You might also consider archiving this historical data to Amazon Simple Storage Service (Amazon S3).

When the total size of the item collection drops below 10 GB, you can once again add items with the same partition key value.

We recommend as a best practice that you instrument your application to monitor the sizes of your item collections. One way to do so is to set the `ReturnItemCollectionMetrics` parameter to `SIZE` whenever you use `BatchWriteItem`, `DeleteItem`, `PutItem`, or `UpdateItem`. Your application should examine the `ReturnItemCollectionMetrics` object in the output and log an error message whenever an item collection exceeds a user-defined limit (8 GB, for example). Setting a limit that is less than 10 GB would provide an early warning system so you know that an item collection is approaching the limit in time to do something about it.

Item collections and partitions

In a table with one or more local secondary indexes, each item collection is stored in one partition. The total size of such an item collection is limited to the capability of that partition: 10 GB. For an application where the data model includes item collections which are unbounded in size, or where you might reasonably expect some item collections to grow beyond 10 GB in the future, you should consider using a global secondary index instead.

You should design your applications so that table data is evenly distributed across distinct partition key values. For tables with local secondary indexes, your applications should not create "hot spots" of read and write activity within a single item collection on a single partition.

Working with Local Secondary Indexes: Java

You can use the AWS SDK for Java Document API to create an Amazon DynamoDB table with one or more local secondary indexes, describe the indexes on the table, and perform queries using the indexes.

The following are the common steps for table operations using the AWS SDK for Java Document API.

1. Create an instance of the `DynamoDB` class.
2. Provide the required and optional parameters for the operation by creating the corresponding request objects.
3. Call the appropriate method provided by the client that you created in the preceding step.

Topics

- Create a table with a Local Secondary Index
- Describe a table with a Local Secondary Index
- Query a Local Secondary Index
- Example: Local Secondary Indexes using the Java document API

Create a table with a Local Secondary Index

Local secondary indexes must be created at the same time you create a table. To do this, use the `createTable` method and provide your specifications for one or more local secondary

indexes. The following Java code example creates a table to hold information about songs in a music collection. The partition key is `Artist` and the sort key is `SongTitle`. A secondary index, `AlbumTitleIndex`, facilitates queries by album title.

The following are the steps to create a table with a local secondary index, using the DynamoDB document API.

1. Create an instance of the `DynamoDB` class.
2. Create an instance of the `CreateTableRequest` class to provide the request information.

   You must provide the table name, its primary key, and the provisioned throughput values. For the local secondary index, you must provide the index name, the name and data type for the index sort key, the key schema for the index, and the attribute projection.
3. Call the `createTable` method by providing the request object as a parameter.

The following Java code example demonstrates the preceding steps. The code creates a table (`Music`) with a secondary index on the `AlbumTitle` attribute. The table partition key and sort key, plus the index sort key, are the only attributes projected into the index.

```
AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();
DynamoDB dynamoDB = new DynamoDB(client);

String tableName = "Music";

CreateTableRequest createTableRequest = new
 CreateTableRequest().withTableName(tableName);

//ProvisionedThroughput
createTableRequest.setProvisionedThroughput(new
 ProvisionedThroughput().withReadCapacityUnits((long)5).withWriteCapacityUnits((long)5));

//AttributeDefinitions
ArrayList<AttributeDefinition> attributeDefinitions= new
 ArrayList<AttributeDefinition>();
attributeDefinitions.add(new
 AttributeDefinition().withAttributeName("Artist").withAttributeType("S"));
attributeDefinitions.add(new
 AttributeDefinition().withAttributeName("SongTitle").withAttributeType("S"));
attributeDefinitions.add(new
 AttributeDefinition().withAttributeName("AlbumTitle").withAttributeType("S"));
```

```
createTableRequest.setAttributeDefinitions(attributeDefinitions);

//KeySchema
ArrayList<KeySchemaElement> tableKeySchema = new ArrayList<KeySchemaElement>();
tableKeySchema.add(new
 KeySchemaElement().withAttributeName("Artist").withKeyType(KeyType.HASH));  //
Partition key
tableKeySchema.add(new
 KeySchemaElement().withAttributeName("SongTitle").withKeyType(KeyType.RANGE));  //Sort
 key

createTableRequest.setKeySchema(tableKeySchema);

ArrayList<KeySchemaElement> indexKeySchema = new ArrayList<KeySchemaElement>();
indexKeySchema.add(new
 KeySchemaElement().withAttributeName("Artist").withKeyType(KeyType.HASH));  //
Partition key
indexKeySchema.add(new
 KeySchemaElement().withAttributeName("AlbumTitle").withKeyType(KeyType.RANGE));  //
Sort key

Projection projection = new Projection().withProjectionType(ProjectionType.INCLUDE);
ArrayList<String> nonKeyAttributes = new ArrayList<String>();
nonKeyAttributes.add("Genre");
nonKeyAttributes.add("Year");
projection.setNonKeyAttributes(nonKeyAttributes);

LocalSecondaryIndex localSecondaryIndex = new LocalSecondaryIndex()

 .withIndexName("AlbumTitleIndex").withKeySchema(indexKeySchema).withProjection(projection);

ArrayList<LocalSecondaryIndex> localSecondaryIndexes = new
 ArrayList<LocalSecondaryIndex>();
localSecondaryIndexes.add(localSecondaryIndex);
createTableRequest.setLocalSecondaryIndexes(localSecondaryIndexes);

Table table = dynamoDB.createTable(createTableRequest);
System.out.println(table.getDescription());
```

You must wait until DynamoDB creates the table and sets the table status to `ACTIVE`. After that, you can begin putting data items into the table.

Describe a table with a Local Secondary Index

To get information about local secondary indexes on a table, use the `describeTable` method. For each index, you can access its name, key schema, and projected attributes.

The following are the steps to access local secondary index information of a table using the AWS SDK for Java Document API.

1. Create an instance of the `DynamoDB` class.
2. Create an instance of the `Table` class. You must provide the table name.
3. Call the `describeTable` method on the `Table` object.

The following Java code example demonstrates the preceding steps.

Example

```
AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();
DynamoDB dynamoDB = new DynamoDB(client);

String tableName = "Music";

Table table = dynamoDB.getTable(tableName);

TableDescription tableDescription = table.describe();

List<LocalSecondaryIndexDescription> localSecondaryIndexes
    = tableDescription.getLocalSecondaryIndexes();

// This code snippet will work for multiple indexes, even though
// there is only one index in this example.

Iterator<LocalSecondaryIndexDescription> lsiIter = localSecondaryIndexes.iterator();
while (lsiIter.hasNext()) {

    LocalSecondaryIndexDescription lsiDescription = lsiIter.next();
    System.out.println("Info for index " + lsiDescription.getIndexName() + ":");
    Iterator<KeySchemaElement> kseIter = lsiDescription.getKeySchema().iterator();
    while (kseIter.hasNext()) {
        KeySchemaElement kse = kseIter.next();
        System.out.printf("\t%s: %s\n", kse.getAttributeName(), kse.getKeyType());
    }
    Projection projection = lsiDescription.getProjection();
```

```
    System.out.println("\tThe projection type is: " + projection.getProjectionType());
    if (projection.getProjectionType().toString().equals("INCLUDE")) {
        System.out.println("\t\tThe non-key projected attributes are: " +
 projection.getNonKeyAttributes());
    }
}
```

Query a Local Secondary Index

You can use the `Query` operation on a local secondary index in much the same way that you `Query` a table. You must specify the index name, the query criteria for the index sort key, and the attributes that you want to return. In this example, the index is `AlbumTitleIndex` and the index sort key is `AlbumTitle`.

The only attributes returned are those that have been projected into the index. You could modify this query to select non-key attributes too, but this would require table fetch activity that is relatively expensive. For more information about table fetches, see Attribute projections.

The following are the steps to query a local secondary index using the AWS SDK for Java Document API.

1. Create an instance of the `DynamoDB` class.
2. Create an instance of the `Table` class. You must provide the table name.
3. Create an instance of the `Index` class. You must provide the index name.
4. Call the `query` method of the `Index` class.

The following Java code example demonstrates the preceding steps.

Example

```
AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();
DynamoDB dynamoDB = new DynamoDB(client);

String tableName = "Music";

Table table = dynamoDB.getTable(tableName);
Index index = table.getIndex("AlbumTitleIndex");

QuerySpec spec = new QuerySpec()
    .withKeyConditionExpression("Artist = :v_artist and AlbumTitle = :v_title")
```

```
    .withValueMap(new ValueMap()
        .withString(":v_artist", "Acme Band")
        .withString(":v_title", "Songs About Life"));

ItemCollection<QueryOutcome> items = index.query(spec);

Iterator<Item> itemsIter = items.iterator();

while (itemsIter.hasNext()) {
    Item item = itemsIter.next();
    System.out.println(item.toJSONPretty());
}
```

Consistent reads on a Local Secondary Index

Unlike global secondary indexes, which only support eventually consistent reads, a local secondary index supports both eventually consistent and strongly consistent reads. A strongly consistent read from a local secondary index always returns the latest updated values. If the query needs to fetch additional attributes from the base table, those fetched attributes are also consistent with respect to the index.

By default, `Query` uses eventually consistent reads. To request a strongly consistent read, set `ConsistentRead` to `true` in the `QuerySpec`. The following example queries `AlbumTitleIndex` using a strongly consistent read:

Example

```
QuerySpec spec = new QuerySpec()
    .withKeyConditionExpression("Artist = :v_artist and AlbumTitle = :v_title")
    .withValueMap(new ValueMap()
        .withString(":v_artist", "Acme Band")
        .withString(":v_title", "Songs About Life"))
    .withConsistentRead(true);
```

> **Note**
>
> A strongly consistent read consumes one read capacity unit per 4 KB of data returned (rounded up), whereas an eventually consistent read consumes half of that. For example, a strongly consistent read that returns 9 KB of data consumes 3 read capacity units (9 KB / 4 KB = 2.25, rounded up to 3), while the same query using an eventually consistent read consumes 1.5 read capacity units. If your application can tolerate reading data that might

be slightly stale, use eventually consistent reads to reduce your read capacity usage. For more information, see Read capacity units.

Example: Local Secondary Indexes using the Java document API

The following Java code example shows how to work with local secondary indexes in Amazon DynamoDB. The example creates a table named `CustomerOrders` with a partition key of `CustomerId` and a sort key of `OrderId`. There are two local secondary indexes on this table:

- `OrderCreationDateIndex` — The sort key is `OrderCreationDate`, and the following attributes are projected into the index:
  - `ProductCategory`
  - `ProductName`
  - `OrderStatus`
  - `ShipmentTrackingId`
- `IsOpenIndex` — The sort key is `IsOpen`, and all of the table attributes are projected into the index.

After the `CustomerOrders` table is created, the program loads the table with data representing customer orders. It then queries the data using the local secondary indexes. Finally, the program deletes the `CustomerOrders` table.

For step-by-step instructions for testing the following sample, see Java code examples.

Example

```
package com.example.dynamodb;

import software.amazon.awssdk.core.waiters.WaiterResponse;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.*;
import software.amazon.awssdk.services.dynamodb.waiters.DynamoDbWaiter;

import java.util.HashMap;
import java.util.Map;

public class DocumentAPILocalSecondaryIndexExample {
```

```
    static DynamoDbClient client = DynamoDbClient.create();
    public static String tableName = "CustomerOrders";

    public static void main(String[] args) {
        createTable();
        loadData();
        query(null);
        query("IsOpenIndex");
        query("OrderCreationDateIndex");
        deleteTable(tableName);
    }

    public static void createTable() {
        CreateTableRequest request = CreateTableRequest.builder()
            .tableName(tableName)
            .provisionedThroughput(ProvisionedThroughput.builder()
                .readCapacityUnits(1L)
                .writeCapacityUnits(1L)
                .build())
            .attributeDefinitions(

AttributeDefinition.builder().attributeName("CustomerId").attributeType(ScalarAttributeType.S)

AttributeDefinition.builder().attributeName("OrderId").attributeType(ScalarAttributeType.N).bu

AttributeDefinition.builder().attributeName("OrderCreationDate").attributeType(ScalarAttribute

AttributeDefinition.builder().attributeName("IsOpen").attributeType(ScalarAttributeType.N).bui
            .keySchema(

KeySchemaElement.builder().attributeName("CustomerId").keyType(KeyType.HASH).build(),

KeySchemaElement.builder().attributeName("OrderId").keyType(KeyType.RANGE).build())
            .localSecondaryIndexes(
                LocalSecondaryIndex.builder()
                    .indexName("OrderCreationDateIndex")
                    .keySchema(

KeySchemaElement.builder().attributeName("CustomerId").keyType(KeyType.HASH).build(),

KeySchemaElement.builder().attributeName("OrderCreationDate").keyType(KeyType.RANGE).build())
                    .projection(Projection.builder()
                        .projectionType(ProjectionType.INCLUDE)
```

```
                        .nonKeyAttributes("ProductCategory", "ProductName")
                        .build())
                    .build(),
                LocalSecondaryIndex.builder()
                    .indexName("IsOpenIndex")
                    .keySchema(

 KeySchemaElement.builder().attributeName("CustomerId").keyType(KeyType.HASH).build(),

 KeySchemaElement.builder().attributeName("IsOpen").keyType(KeyType.RANGE).build())
                    .projection(Projection.builder()
                        .projectionType(ProjectionType.ALL)
                        .build())
                    .build())
            .build();

        System.out.println("Creating table " + tableName + "...");
        client.createTable(request);

        try (DynamoDbWaiter waiter = client.waiter()) {
            WaiterResponse<DescribeTableResponse> response =
 waiter.waitUntilTableExists(r -> r.tableName(tableName));
            response.matched().response().ifPresent(System.out::println);
        }
    }

    public static void query(String indexName) {

 System.out.println("\n***********************************************************\n");
        System.out.println("Querying table " + tableName + "...");

        if ("IsOpenIndex".equals(indexName)) {
            System.out.println("\nUsing index: '" + indexName + "': Bob's orders that
 are open.");
            System.out.println("Only a user-specified list of attributes are returned
\n");

            Map<String, AttributeValue> values = new HashMap<>();
            values.put(":v_custid",
 AttributeValue.builder().s("bob@example.com").build());
            values.put(":v_isopen", AttributeValue.builder().n("1").build());

            QueryRequest request = QueryRequest.builder()
                .tableName(tableName)
```

```
                .indexName(indexName)
                .keyConditionExpression("CustomerId = :v_custid and IsOpen
= :v_isopen")
                .expressionAttributeValues(values)
                .projectionExpression("OrderCreationDate, ProductCategory, ProductName,
OrderStatus")
                .build();

            System.out.println("Query: printing results...");
            client.query(request).items().forEach(System.out::println);

        } else if ("OrderCreationDateIndex".equals(indexName)) {
            System.out.println("\nUsing index: '" + indexName + "': Bob's orders that
were placed after 01/31/2015.");
            System.out.println("Only the projected attributes are returned\n");

            Map<String, AttributeValue> values = new HashMap<>();
            values.put(":v_custid",
AttributeValue.builder().s("bob@example.com").build());
            values.put(":v_orddate", AttributeValue.builder().n("20150131").build());

            QueryRequest request = QueryRequest.builder()
                .tableName(tableName)
                .indexName(indexName)
                .keyConditionExpression("CustomerId = :v_custid and OrderCreationDate
>= :v_orddate")
                .expressionAttributeValues(values)
                .select(Select.ALL_PROJECTED_ATTRIBUTES)
                .build();

            System.out.println("Query: printing results...");
            client.query(request).items().forEach(System.out::println);

        } else {
            System.out.println("\nNo index: All of Bob's orders, by OrderId:\n");

            Map<String, AttributeValue> values = new HashMap<>();
            values.put(":v_custid",
AttributeValue.builder().s("bob@example.com").build());

            QueryRequest request = QueryRequest.builder()
                .tableName(tableName)
                .keyConditionExpression("CustomerId = :v_custid")
                .expressionAttributeValues(values)
```

```
                .build();

        System.out.println("Query: printing results...");
        client.query(request).items().forEach(System.out::println);
    }
}

public static void deleteTable(String tableName) {
    System.out.println("Deleting table " + tableName + "...");
    client.deleteTable(DeleteTableRequest.builder().tableName(tableName).build());

    try (DynamoDbWaiter waiter = client.waiter()) {
        waiter.waitUntilTableNotExists(r -> r.tableName(tableName));
    }
}

public static void loadData() {
    System.out.println("Loading data into table " + tableName + "...");

    putItem(Map.of(
        "CustomerId", AttributeValue.builder().s("alice@example.com").build(),
        "OrderId", AttributeValue.builder().n("1").build(),
        "IsOpen", AttributeValue.builder().n("1").build(),
        "OrderCreationDate", AttributeValue.builder().n("20150101").build(),
        "ProductCategory", AttributeValue.builder().s("Book").build(),
        "ProductName", AttributeValue.builder().s("The Great Outdoors").build(),
        "OrderStatus", AttributeValue.builder().s("PACKING ITEMS").build()));

    putItem(Map.of(
        "CustomerId", AttributeValue.builder().s("alice@example.com").build(),
        "OrderId", AttributeValue.builder().n("2").build(),
        "IsOpen", AttributeValue.builder().n("1").build(),
        "OrderCreationDate", AttributeValue.builder().n("20150221").build(),
        "ProductCategory", AttributeValue.builder().s("Bike").build(),
        "ProductName", AttributeValue.builder().s("Super Mountain").build(),
        "OrderStatus", AttributeValue.builder().s("ORDER RECEIVED").build()));

    putItem(Map.of(
        "CustomerId", AttributeValue.builder().s("alice@example.com").build(),
        "OrderId", AttributeValue.builder().n("3").build(),
        "OrderCreationDate", AttributeValue.builder().n("20150304").build(),
        "ProductCategory", AttributeValue.builder().s("Music").build(),
        "ProductName", AttributeValue.builder().s("A Quiet Interlude").build(),
        "OrderStatus", AttributeValue.builder().s("IN TRANSIT").build(),
```

```
            "ShipmentTrackingId", AttributeValue.builder().s("176493").build()));

        putItem(Map.of(
            "CustomerId", AttributeValue.builder().s("bob@example.com").build(),
            "OrderId", AttributeValue.builder().n("1").build(),
            "OrderCreationDate", AttributeValue.builder().n("20150111").build(),
            "ProductCategory", AttributeValue.builder().s("Movie").build(),
            "ProductName", AttributeValue.builder().s("Calm Before The Storm").build(),
            "OrderStatus", AttributeValue.builder().s("SHIPPING DELAY").build(),
            "ShipmentTrackingId", AttributeValue.builder().s("859323").build()));

        putItem(Map.of(
            "CustomerId", AttributeValue.builder().s("bob@example.com").build(),
            "OrderId", AttributeValue.builder().n("2").build(),
            "OrderCreationDate", AttributeValue.builder().n("20150124").build(),
            "ProductCategory", AttributeValue.builder().s("Music").build(),
            "ProductName", AttributeValue.builder().s("E-Z Listening").build(),
            "OrderStatus", AttributeValue.builder().s("DELIVERED").build(),
            "ShipmentTrackingId", AttributeValue.builder().s("756943").build()));

        putItem(Map.of(
            "CustomerId", AttributeValue.builder().s("bob@example.com").build(),
            "OrderId", AttributeValue.builder().n("3").build(),
            "OrderCreationDate", AttributeValue.builder().n("20150221").build(),
            "ProductCategory", AttributeValue.builder().s("Music").build(),
            "ProductName", AttributeValue.builder().s("Symphony 9").build(),
            "OrderStatus", AttributeValue.builder().s("DELIVERED").build(),
            "ShipmentTrackingId", AttributeValue.builder().s("645193").build()));

        putItem(Map.of(
            "CustomerId", AttributeValue.builder().s("bob@example.com").build(),
            "OrderId", AttributeValue.builder().n("4").build(),
            "IsOpen", AttributeValue.builder().n("1").build(),
            "OrderCreationDate", AttributeValue.builder().n("20150222").build(),
            "ProductCategory", AttributeValue.builder().s("Hardware").build(),
            "ProductName", AttributeValue.builder().s("Extra Heavy Hammer").build(),
            "OrderStatus", AttributeValue.builder().s("PACKING ITEMS").build()));

        putItem(Map.of(
            "CustomerId", AttributeValue.builder().s("bob@example.com").build(),
            "OrderId", AttributeValue.builder().n("5").build(),
            "OrderCreationDate", AttributeValue.builder().n("20150309").build(),
            "ProductCategory", AttributeValue.builder().s("Book").build(),
            "ProductName", AttributeValue.builder().s("How To Cook").build(),
```

```
            "OrderStatus", AttributeValue.builder().s("IN TRANSIT").build(),
            "ShipmentTrackingId", AttributeValue.builder().s("440185").build()));

        putItem(Map.of(
            "CustomerId", AttributeValue.builder().s("bob@example.com").build(),
            "OrderId", AttributeValue.builder().n("6").build(),
            "OrderCreationDate", AttributeValue.builder().n("20150318").build(),
            "ProductCategory", AttributeValue.builder().s("Luggage").build(),
            "ProductName", AttributeValue.builder().s("Really Big Suitcase").build(),
            "OrderStatus", AttributeValue.builder().s("DELIVERED").build(),
            "ShipmentTrackingId", AttributeValue.builder().s("893927").build()));

        putItem(Map.of(
            "CustomerId", AttributeValue.builder().s("bob@example.com").build(),
            "OrderId", AttributeValue.builder().n("7").build(),
            "OrderCreationDate", AttributeValue.builder().n("20150324").build(),
            "ProductCategory", AttributeValue.builder().s("Golf").build(),
            "ProductName", AttributeValue.builder().s("PGA Pro II").build(),
            "OrderStatus", AttributeValue.builder().s("OUT FOR DELIVERY").build(),
            "ShipmentTrackingId", AttributeValue.builder().s("383283").build()));
    }

    private static void putItem(Map<String, AttributeValue> item) {

 client.putItem(PutItemRequest.builder().tableName(tableName).item(item).build());
    }
}
```

Working with Local Secondary Indexes: .NET

Topics

- Create a table with a Local Secondary Index
- Describe a table with a Local Secondary Index
- Query a Local Secondary Index
- Example: Local Secondary Indexes using the AWS SDK for .NET low-level API

You can use the AWS SDK for .NET low-level API to create an Amazon DynamoDB table with one or more local secondary indexes, describe the indexes on the table, and perform queries using the

indexes. These operations map to the corresponding low-level DynamoDB API actions. For more information, see .NET code examples.

The following are the common steps for table operations using the .NET low-level API.

1. Create an instance of the `AmazonDynamoDBClient` class.
2. Provide the required and optional parameters for the operation by creating the corresponding request objects.

   For example, create a `CreateTableRequest` object to create a table and an `QueryRequest` object to query a table or an index.
3. Run the appropriate method provided by the client that you created in the preceding step.

Create a table with a Local Secondary Index

Local secondary indexes must be created at the same time that you create a table. To do this, use `CreateTable` and provide your specifications for one or more local secondary indexes. The following C# code example creates a table to hold information about songs in a music collection. The partition key is `Artist` and the sort key is `SongTitle`. A secondary index, `AlbumTitleIndex`, facilitates queries by album title.

The following are the steps to create a table with a local secondary index, using the .NET low-level API.

1. Create an instance of the `AmazonDynamoDBClient` class.
2. Create an instance of the `CreateTableRequest` class to provide the request information.

   You must provide the table name, its primary key, and the provisioned throughput values. For the local secondary index, you must provide the index name, the name and data type of the index sort key, the key schema for the index, and the attribute projection.
3. Run the `CreateTable` method by providing the request object as a parameter.

The following C# code example demonstrates the preceding steps. The code creates a table (`Music`) with a secondary index on the `AlbumTitle` attribute. The table partition key and sort key, plus the index sort key, are the only attributes projected into the index.

```
AmazonDynamoDBClient client = new AmazonDynamoDBClient();
string tableName = "Music";
```

```
CreateTableRequest createTableRequest = new CreateTableRequest()
{
    TableName = tableName
};

//ProvisionedThroughput
createTableRequest.ProvisionedThroughput = new ProvisionedThroughput()
{
    ReadCapacityUnits = (long)5,
    WriteCapacityUnits = (long)5
};

//AttributeDefinitions
List<AttributeDefinition> attributeDefinitions = new List<AttributeDefinition>();

attributeDefinitions.Add(new AttributeDefinition()
{
    AttributeName = "Artist",
    AttributeType = "S"
});

attributeDefinitions.Add(new AttributeDefinition()
 {
     AttributeName = "SongTitle",
     AttributeType = "S"
 });

attributeDefinitions.Add(new AttributeDefinition()
 {
     AttributeName = "AlbumTitle",
     AttributeType = "S"
 });

createTableRequest.AttributeDefinitions = attributeDefinitions;

//KeySchema
List<KeySchemaElement> tableKeySchema = new List<KeySchemaElement>();

tableKeySchema.Add(new KeySchemaElement() { AttributeName = "Artist", KeyType =
 "HASH" });  //Partition key
tableKeySchema.Add(new KeySchemaElement() { AttributeName = "SongTitle", KeyType =
 "RANGE" });  //Sort key
```

```
createTableRequest.KeySchema = tableKeySchema;

List<KeySchemaElement> indexKeySchema = new List<KeySchemaElement>();
indexKeySchema.Add(new KeySchemaElement() { AttributeName = "Artist", KeyType =
 "HASH" });  //Partition key
indexKeySchema.Add(new KeySchemaElement() { AttributeName = "AlbumTitle", KeyType =
 "RANGE" });  //Sort key

Projection projection = new Projection() { ProjectionType = "INCLUDE" };

List<string> nonKeyAttributes = new List<string>();
nonKeyAttributes.Add("Genre");
nonKeyAttributes.Add("Year");
projection.NonKeyAttributes = nonKeyAttributes;

LocalSecondaryIndex localSecondaryIndex = new LocalSecondaryIndex()
{
    IndexName = "AlbumTitleIndex",
    KeySchema = indexKeySchema,
    Projection = projection
};

List<LocalSecondaryIndex> localSecondaryIndexes = new List<LocalSecondaryIndex>();
localSecondaryIndexes.Add(localSecondaryIndex);
createTableRequest.LocalSecondaryIndexes = localSecondaryIndexes;

CreateTableResponse result = client.CreateTable(createTableRequest);
Console.WriteLine(result.CreateTableResult.TableDescription.TableName);
Console.WriteLine(result.CreateTableResult.TableDescription.TableStatus);
```

You must wait until DynamoDB creates the table and sets the table status to `ACTIVE`. After that, you can begin putting data items into the table.

Describe a table with a Local Secondary Index

To get information about local secondary indexes on a table, use the `DescribeTable` API. For each index, you can access its name, key schema, and projected attributes.

The following are the steps to access local secondary index information a table using the .NET low-level API.

1. Create an instance of the `AmazonDynamoDBClient` class.

2. Create an instance of the `DescribeTableRequest` class to provide the request information. You must provide the table name.
3. Run the `describeTable` method by providing the request object as a parameter.
4.

The following C# code example demonstrates the preceding steps.

Example

```
AmazonDynamoDBClient client = new AmazonDynamoDBClient();
string tableName = "Music";

DescribeTableResponse response = client.DescribeTable(new DescribeTableRequest()
 { TableName = tableName });
List<LocalSecondaryIndexDescription> localSecondaryIndexes =
    response.DescribeTableResult.Table.LocalSecondaryIndexes;

// This code snippet will work for multiple indexes, even though
// there is only one index in this example.
foreach (LocalSecondaryIndexDescription lsiDescription in localSecondaryIndexes)
{
    Console.WriteLine("Info for index " + lsiDescription.IndexName + ":");

    foreach (KeySchemaElement kse in lsiDescription.KeySchema)
    {
        Console.WriteLine("\t" + kse.AttributeName + ": key type is " + kse.KeyType);
    }

    Projection projection = lsiDescription.Projection;

    Console.WriteLine("\tThe projection type is: " + projection.ProjectionType);

    if (projection.ProjectionType.ToString().Equals("INCLUDE"))
    {
        Console.WriteLine("\t\tThe non-key projected attributes are:");

        foreach (String s in projection.NonKeyAttributes)
        {
            Console.WriteLine("\t\t" + s);
        }

    }
```

```
}
```

Query a Local Secondary Index

You can use `Query` on a local secondary index in much the same way you `Query` a table. You must specify the index name, the query criteria for the index sort key, and the attributes that you want to return. In this example, the index is `AlbumTitleIndex`, and the index sort key is `AlbumTitle`.

The only attributes returned are those that have been projected into the index. You could modify this query to select non-key attributes too, but this would require table fetch activity that is relatively expensive. For more information about table fetches, see Attribute projections

The following are the steps to query a local secondary index using the .NET low-level API.

1. Create an instance of the `AmazonDynamoDBClient` class.
2. Create an instance of the `QueryRequest` class to provide the request information.
3. Run the `query` method by providing the request object as a parameter.

The following C# code example demonstrates the preceding steps.

Example

```
QueryRequest queryRequest = new QueryRequest
{
    TableName = "Music",
    IndexName = "AlbumTitleIndex",
    Select = "ALL_ATTRIBUTES",
    ScanIndexForward = true,
    KeyConditionExpression = "Artist = :v_artist and AlbumTitle = :v_title",
    ExpressionAttributeValues = new Dictionary<string, AttributeValue>()
    {
        {":v_artist",new AttributeValue {S = "Acme Band"}},
        {":v_title",new AttributeValue {S = "Songs About Life"}}
    },
};

QueryResponse response = client.Query(queryRequest);

foreach (var attribs in response.Items)
{
```

```
    foreach (var attrib in attribs)
    {
        Console.WriteLine(attrib.Key + " ---> " + attrib.Value.S);
    }
    Console.WriteLine();
}
```

Example: Local Secondary Indexes using the AWS SDK for .NET low-level API

The following C# code example shows how to work with local secondary indexes in Amazon DynamoDB. The example creates a table named `CustomerOrders` with a partition key of `CustomerId` and a sort key of `OrderId`. There are two local secondary indexes on this table:

- `OrderCreationDateIndex` — The sort key is `OrderCreationDate`, and the following attributes are projected into the index:
  - `ProductCategory`
  - `ProductName`
  - `OrderStatus`
  - `ShipmentTrackingId`
- `IsOpenIndex` — The sort key is `IsOpen`, and all of the table attributes are projected into the index.

After the `CustomerOrders` table is created, the program loads the table with data representing customer orders. It then queries the data using the local secondary indexes. Finally, the program deletes the `CustomerOrders` table.

For step-by-step instructions for testing the following example, see .NET code examples.

Example

```
using System;
using System.Collections.Generic;
using System.Linq;
using Amazon.DynamoDBv2;
using Amazon.DynamoDBv2.DataModel;
using Amazon.DynamoDBv2.DocumentModel;
using Amazon.DynamoDBv2.Model;
using Amazon.Runtime;
using Amazon.SecurityToken;
```

```
namespace com.amazonaws.codesamples
{
    class LowLevelLocalSecondaryIndexExample
    {
        private static AmazonDynamoDBClient client = new AmazonDynamoDBClient();
        private static string tableName = "CustomerOrders";

        static void Main(string[] args)
        {
            try
            {
                CreateTable();
                LoadData();

                Query(null);
                Query("IsOpenIndex");
                Query("OrderCreationDateIndex");

                DeleteTable(tableName);

                Console.WriteLine("To continue, press Enter");
                Console.ReadLine();
            }
            catch (AmazonDynamoDBException e) { Console.WriteLine(e.Message); }
            catch (AmazonServiceException e) { Console.WriteLine(e.Message); }
            catch (Exception e) { Console.WriteLine(e.Message); }
        }

        private static void CreateTable()
        {
            var createTableRequest =
                new CreateTableRequest()
                {
                    TableName = tableName,
                    ProvisionedThroughput =
                    new ProvisionedThroughput()
                    {
                        ReadCapacityUnits = (long)1,
                        WriteCapacityUnits = (long)1
                    }
                };

            var attributeDefinitions = new List<AttributeDefinition>()
        {
```

```
            // Attribute definitions for table primary key
            { new AttributeDefinition() {
                  AttributeName = "CustomerId", AttributeType = "S"
              } },
            { new AttributeDefinition() {
                  AttributeName = "OrderId", AttributeType = "N"
              } },
            // Attribute definitions for index primary key
            { new AttributeDefinition() {
                  AttributeName = "OrderCreationDate", AttributeType = "N"
              } },
            { new AttributeDefinition() {
                  AttributeName = "IsOpen", AttributeType = "N"
              }}
        };

            createTableRequest.AttributeDefinitions = attributeDefinitions;

            // Key schema for table
            var tableKeySchema = new List<KeySchemaElement>()
        {
            { new KeySchemaElement() {
                  AttributeName = "CustomerId", KeyType = "HASH"
              } },                                                  //Partition key
            { new KeySchemaElement() {
                  AttributeName = "OrderId", KeyType = "RANGE"
              } }                                                  //Sort key
        };

            createTableRequest.KeySchema = tableKeySchema;

            var localSecondaryIndexes = new List<LocalSecondaryIndex>();

            // OrderCreationDateIndex
            LocalSecondaryIndex orderCreationDateIndex = new LocalSecondaryIndex()
            {
                IndexName = "OrderCreationDateIndex"
            };

            // Key schema for OrderCreationDateIndex
            var indexKeySchema = new List<KeySchemaElement>()
        {
            { new KeySchemaElement() {
                  AttributeName = "CustomerId", KeyType = "HASH"
```

```
            } },                                                    //Partition key
        { new KeySchemaElement() {
              AttributeName = "OrderCreationDate", KeyType = "RANGE"
            } }                                                                 //Sort key
    };

        orderCreationDateIndex.KeySchema = indexKeySchema;

        // Projection (with list of projected attributes) for
        // OrderCreationDateIndex
        var projection = new Projection()
        {
            ProjectionType = "INCLUDE"
        };

        var nonKeyAttributes = new List<string>()
    {
        "ProductCategory",
        "ProductName"
    };
        projection.NonKeyAttributes = nonKeyAttributes;

        orderCreationDateIndex.Projection = projection;

        localSecondaryIndexes.Add(orderCreationDateIndex);

        // IsOpenIndex
        LocalSecondaryIndex isOpenIndex
            = new LocalSecondaryIndex()
            {
                IndexName = "IsOpenIndex"
            };

        // Key schema for IsOpenIndex
        indexKeySchema = new List<KeySchemaElement>()
    {
        { new KeySchemaElement() {
              AttributeName = "CustomerId", KeyType = "HASH"
          }},                                                       //Partition key
        { new KeySchemaElement() {
              AttributeName = "IsOpen", KeyType = "RANGE"
          }}                                                  //Sort key
    };
```

```
            // Projection (all attributes) for IsOpenIndex
            projection = new Projection()
            {
                ProjectionType = "ALL"
            };

            isOpenIndex.KeySchema = indexKeySchema;
            isOpenIndex.Projection = projection;

            localSecondaryIndexes.Add(isOpenIndex);

            // Add index definitions to CreateTable request
            createTableRequest.LocalSecondaryIndexes = localSecondaryIndexes;

            Console.WriteLine("Creating table " + tableName + "...");
            client.CreateTable(createTableRequest);
            WaitUntilTableReady(tableName);
        }

        public static void Query(string indexName)
        {

Console.WriteLine("\n*****************************************************\n");
            Console.WriteLine("Querying table " + tableName + "...");

            QueryRequest queryRequest = new QueryRequest()
            {
                TableName = tableName,
                ConsistentRead = true,
                ScanIndexForward = true,
                ReturnConsumedCapacity = "TOTAL"
            };


            String keyConditionExpression = "CustomerId = :v_customerId";
            Dictionary<string, AttributeValue> expressionAttributeValues = new
Dictionary<string, AttributeValue> {
            {":v_customerId", new AttributeValue {
                 S = "bob@example.com"
             }}
        };


            if (indexName == "IsOpenIndex")
```

```
            {
                Console.WriteLine("\nUsing index: '" + indexName
                          + "': Bob's orders that are open.");
                Console.WriteLine("Only a user-specified list of attributes are
returned\n");
                queryRequest.IndexName = indexName;

                keyConditionExpression += " and IsOpen = :v_isOpen";
                expressionAttributeValues.Add(":v_isOpen", new AttributeValue
                {
                    N = "1"
                });

                // ProjectionExpression
                queryRequest.ProjectionExpression = "OrderCreationDate,
ProductCategory, ProductName, OrderStatus";
            }
            else if (indexName == "OrderCreationDateIndex")
            {
                Console.WriteLine("\nUsing index: '" + indexName
                          + "': Bob's orders that were placed after 01/31/2013.");
                Console.WriteLine("Only the projected attributes are returned\n");
                queryRequest.IndexName = indexName;

                keyConditionExpression += " and OrderCreationDate > :v_Date";
                expressionAttributeValues.Add(":v_Date", new AttributeValue
                {
                    N = "20130131"
                });

                // Select
                queryRequest.Select = "ALL_PROJECTED_ATTRIBUTES";
            }
            else
            {
                Console.WriteLine("\nNo index: All of Bob's orders, by OrderId:\n");
            }
            queryRequest.KeyConditionExpression = keyConditionExpression;
            queryRequest.ExpressionAttributeValues = expressionAttributeValues;

            var result = client.Query(queryRequest);
            var items = result.Items;
            foreach (var currentItem in items)
            {
```

```
                foreach (string attr in currentItem.Keys)
                {
                    if (attr == "OrderId" || attr == "IsOpen"
                        || attr == "OrderCreationDate")
                    {
                        Console.WriteLine(attr + "---> " + currentItem[attr].N);
                    }
                    else
                    {
                        Console.WriteLine(attr + "---> " + currentItem[attr].S);
                    }
                }
                Console.WriteLine();
            }
            Console.WriteLine("\nConsumed capacity: " +
result.ConsumedCapacity.CapacityUnits + "\n");
        }

        private static void DeleteTable(string tableName)
        {
            Console.WriteLine("Deleting table " + tableName + "...");
            client.DeleteTable(new DeleteTableRequest()
            {
                TableName = tableName
            });
            WaitForTableToBeDeleted(tableName);
        }

        public static void LoadData()
        {
            Console.WriteLine("Loading data into table " + tableName + "...");

            Dictionary<string, AttributeValue> item = new Dictionary<string,
AttributeValue>();

            item["CustomerId"] = new AttributeValue
            {
                S = "alice@example.com"
            };
            item["OrderId"] = new AttributeValue
            {
                N = "1"
            };
            item["IsOpen"] = new AttributeValue
```

```
            {
                N = "1"
            };
            item["OrderCreationDate"] = new AttributeValue
            {
                N = "20130101"
            };
            item["ProductCategory"] = new AttributeValue
            {
                S = "Book"
            };
            item["ProductName"] = new AttributeValue
            {
                S = "The Great Outdoors"
            };
            item["OrderStatus"] = new AttributeValue
            {
                S = "PACKING ITEMS"
            };
            /* no ShipmentTrackingId attribute */
            PutItemRequest putItemRequest = new PutItemRequest
            {
                TableName = tableName,
                Item = item,
                ReturnItemCollectionMetrics = "SIZE"
            };
            client.PutItem(putItemRequest);

            item = new Dictionary<string, AttributeValue>();
            item["CustomerId"] = new AttributeValue
            {
                S = "alice@example.com"
            };
            item["OrderId"] = new AttributeValue
            {
                N = "2"
            };
            item["IsOpen"] = new AttributeValue
            {
                N = "1"
            };
            item["OrderCreationDate"] = new AttributeValue
            {
                N = "20130221"
```

```
            };
            item["ProductCategory"] = new AttributeValue
            {
                S = "Bike"
            };
            item["ProductName"] = new AttributeValue
            {
                S = "Super Mountain"
            };
            item["OrderStatus"] = new AttributeValue
            {
                S = "ORDER RECEIVED"
            };
            /* no ShipmentTrackingId attribute */
            putItemRequest = new PutItemRequest
            {
                TableName = tableName,
                Item = item,
                ReturnItemCollectionMetrics = "SIZE"
            };
            client.PutItem(putItemRequest);

            item = new Dictionary<string, AttributeValue>();
            item["CustomerId"] = new AttributeValue
            {
                S = "alice@example.com"
            };
            item["OrderId"] = new AttributeValue
            {
                N = "3"
            };
            /* no IsOpen attribute */
            item["OrderCreationDate"] = new AttributeValue
            {
                N = "20130304"
            };
            item["ProductCategory"] = new AttributeValue
            {
                S = "Music"
            };
            item["ProductName"] = new AttributeValue
            {
                S = "A Quiet Interlude"
            };
```

```
            item["OrderStatus"] = new AttributeValue
            {
                S = "IN TRANSIT"
            };
            item["ShipmentTrackingId"] = new AttributeValue
            {
                S = "176493"
            };
            putItemRequest = new PutItemRequest
            {
                TableName = tableName,
                Item = item,
                ReturnItemCollectionMetrics = "SIZE"
            };
            client.PutItem(putItemRequest);

            item = new Dictionary<string, AttributeValue>();
            item["CustomerId"] = new AttributeValue
            {
                S = "bob@example.com"
            };
            item["OrderId"] = new AttributeValue
            {
                N = "1"
            };
            /* no IsOpen attribute */
            item["OrderCreationDate"] = new AttributeValue
            {
                N = "20130111"
            };
            item["ProductCategory"] = new AttributeValue
            {
                S = "Movie"
            };
            item["ProductName"] = new AttributeValue
            {
                S = "Calm Before The Storm"
            };
            item["OrderStatus"] = new AttributeValue
            {
                S = "SHIPPING DELAY"
            };
            item["ShipmentTrackingId"] = new AttributeValue
            {
```

```
            S = "859323"
        };
        putItemRequest = new PutItemRequest
        {
            TableName = tableName,
            Item = item,
            ReturnItemCollectionMetrics = "SIZE"
        };
        client.PutItem(putItemRequest);

        item = new Dictionary<string, AttributeValue>();
        item["CustomerId"] = new AttributeValue
        {
            S = "bob@example.com"
        };
        item["OrderId"] = new AttributeValue
        {
            N = "2"
        };
        /* no IsOpen attribute */
        item["OrderCreationDate"] = new AttributeValue
        {
            N = "20130124"
        };
        item["ProductCategory"] = new AttributeValue
        {
            S = "Music"
        };
        item["ProductName"] = new AttributeValue
        {
            S = "E-Z Listening"
        };
        item["OrderStatus"] = new AttributeValue
        {
            S = "DELIVERED"
        };
        item["ShipmentTrackingId"] = new AttributeValue
        {
            S = "756943"
        };
        putItemRequest = new PutItemRequest
        {
            TableName = tableName,
            Item = item,
```

```
            ReturnItemCollectionMetrics = "SIZE"
        };
        client.PutItem(putItemRequest);

        item = new Dictionary<string, AttributeValue>();
        item["CustomerId"] = new AttributeValue
        {
            S = "bob@example.com"
        };
        item["OrderId"] = new AttributeValue
        {
            N = "3"
        };
        /* no IsOpen attribute */
        item["OrderCreationDate"] = new AttributeValue
        {
            N = "20130221"
        };
        item["ProductCategory"] = new AttributeValue
        {
            S = "Music"
        };
        item["ProductName"] = new AttributeValue
        {
            S = "Symphony 9"
        };
        item["OrderStatus"] = new AttributeValue
        {
            S = "DELIVERED"
        };
        item["ShipmentTrackingId"] = new AttributeValue
        {
            S = "645193"
        };
        putItemRequest = new PutItemRequest
        {
            TableName = tableName,
            Item = item,
            ReturnItemCollectionMetrics = "SIZE"
        };
        client.PutItem(putItemRequest);

        item = new Dictionary<string, AttributeValue>();
        item["CustomerId"] = new AttributeValue
```

```
            {
                S = "bob@example.com"
            };
            item["OrderId"] = new AttributeValue
            {
                N = "4"
            };
            item["IsOpen"] = new AttributeValue
            {
                N = "1"
            };
            item["OrderCreationDate"] = new AttributeValue
            {
                N = "20130222"
            };
            item["ProductCategory"] = new AttributeValue
            {
                S = "Hardware"
            };
            item["ProductName"] = new AttributeValue
            {
                S = "Extra Heavy Hammer"
            };
            item["OrderStatus"] = new AttributeValue
            {
                S = "PACKING ITEMS"
            };
            /* no ShipmentTrackingId attribute */
            putItemRequest = new PutItemRequest
            {
                TableName = tableName,
                Item = item,
                ReturnItemCollectionMetrics = "SIZE"
            };
            client.PutItem(putItemRequest);

            item = new Dictionary<string, AttributeValue>();
            item["CustomerId"] = new AttributeValue
            {
                S = "bob@example.com"
            };
            item["OrderId"] = new AttributeValue
            {
                N = "5"
```

```
            };
            /* no IsOpen attribute */
            item["OrderCreationDate"] = new AttributeValue
            {
                N = "20130309"
            };
            item["ProductCategory"] = new AttributeValue
            {
                S = "Book"
            };
            item["ProductName"] = new AttributeValue
            {
                S = "How To Cook"
            };
            item["OrderStatus"] = new AttributeValue
            {
                S = "IN TRANSIT"
            };
            item["ShipmentTrackingId"] = new AttributeValue
            {
                S = "440185"
            };
            putItemRequest = new PutItemRequest
            {
                TableName = tableName,
                Item = item,
                ReturnItemCollectionMetrics = "SIZE"
            };
            client.PutItem(putItemRequest);

            item = new Dictionary<string, AttributeValue>();
            item["CustomerId"] = new AttributeValue
            {
                S = "bob@example.com"
            };
            item["OrderId"] = new AttributeValue
            {
                N = "6"
            };
            /* no IsOpen attribute */
            item["OrderCreationDate"] = new AttributeValue
            {
                N = "20130318"
            };
```

```
            item["ProductCategory"] = new AttributeValue
            {
                S = "Luggage"
            };
            item["ProductName"] = new AttributeValue
            {
                S = "Really Big Suitcase"
            };
            item["OrderStatus"] = new AttributeValue
            {
                S = "DELIVERED"
            };
            item["ShipmentTrackingId"] = new AttributeValue
            {
                S = "893927"
            };
            putItemRequest = new PutItemRequest
            {
                TableName = tableName,
                Item = item,
                ReturnItemCollectionMetrics = "SIZE"
            };
            client.PutItem(putItemRequest);

            item = new Dictionary<string, AttributeValue>();
            item["CustomerId"] = new AttributeValue
            {
                S = "bob@example.com"
            };
            item["OrderId"] = new AttributeValue
            {
                N = "7"
            };
            /* no IsOpen attribute */
            item["OrderCreationDate"] = new AttributeValue
            {
                N = "20130324"
            };
            item["ProductCategory"] = new AttributeValue
            {
                S = "Golf"
            };
            item["ProductName"] = new AttributeValue
            {
```

```
                S = "PGA Pro II"
            };
            item["OrderStatus"] = new AttributeValue
            {
                S = "OUT FOR DELIVERY"
            };
            item["ShipmentTrackingId"] = new AttributeValue
            {
                S = "383283"
            };
            putItemRequest = new PutItemRequest
            {
                TableName = tableName,
                Item = item,
                ReturnItemCollectionMetrics = "SIZE"
            };
            client.PutItem(putItemRequest);
        }

        private static void WaitUntilTableReady(string tableName)
        {
            string status = null;
            // Let us wait until table is created. Call DescribeTable.
            do
            {
                System.Threading.Thread.Sleep(5000); // Wait 5 seconds.
                try
                {
                    var res = client.DescribeTable(new DescribeTableRequest
                    {
                        TableName = tableName
                    });

                    Console.WriteLine("Table name: {0}, status: {1}",
                              res.Table.TableName,
                              res.Table.TableStatus);
                    status = res.Table.TableStatus;
                }
                catch (ResourceNotFoundException)
                {
                    // DescribeTable is eventually consistent. So you might
                    // get resource not found. So we handle the potential exception.
                }
            } while (status != "ACTIVE");
```

```
        }

        private static void WaitForTableToBeDeleted(string tableName)
        {
            bool tablePresent = true;

            while (tablePresent)
            {
                System.Threading.Thread.Sleep(5000); // Wait 5 seconds.
                try
                {
                    var res = client.DescribeTable(new DescribeTableRequest
                    {
                        TableName = tableName
                    });

                    Console.WriteLine("Table name: {0}, status: {1}",
                              res.Table.TableName,
                              res.Table.TableStatus);
                }
                catch (ResourceNotFoundException)
                {
                    tablePresent = false;
                }
            }
        }
    }
}
```

Working with Local Secondary Indexes in DynamoDB AWS CLI

You can use the AWS CLI to create an Amazon DynamoDB table with one or more Local Secondary Indexes, describe the indexes on the table, and perform queries using the indexes.

Topics

- Create a table with a Local Secondary Index
- Describe a table with a Local Secondary Index
- Query a Local Secondary Index

Create a table with a Local Secondary Index

Local Secondary Indexes must be created at the same time you create a table. To do this, use the `create-table` parameter and provide your specifications for one or more Local Secondary Indexes. The following example creates a table (`Music`) to hold information about songs in a music collection. The partition key is `Artist` and the sort key is `SongTitle`. A secondary index, `AlbumTitleIndex` on the `AlbumTitle` attribute facilitates queries by album title.

```
aws dynamodb create-table \
    --table-name Music \
    --attribute-definitions AttributeName=Artist,AttributeType=S
 AttributeName=SongTitle,AttributeType=S \
        AttributeName=AlbumTitle,AttributeType=S  \
    --key-schema AttributeName=Artist,KeyType=HASH
 AttributeName=SongTitle,KeyType=RANGE \
    --provisioned-throughput \
        ReadCapacityUnits=10,WriteCapacityUnits=5 \
    --local-secondary-indexes \
        "[{\"IndexName\": \"AlbumTitleIndex\",
        \"KeySchema\":[{\"AttributeName\":\"Artist\",\"KeyType\":\"HASH\"},
                      {\"AttributeName\":\"AlbumTitle\",\"KeyType\":\"RANGE\"}],
        \"Projection\":{\"ProjectionType\":\"INCLUDE\",  \"NonKeyAttributes\":[\"Genre
\", \"Year\"]}}]"
```

You must wait until DynamoDB creates the table and sets the table status to `ACTIVE`. After that, you can begin putting data items into the table. You can use describe-table to determine the status of the table creation.

Describe a table with a Local Secondary Index

To get information about local secondary indexes on a table, use the `describe-table` parameter. For each index, you can access its name, key schema, and projected attributes.

```
aws dynamodb describe-table --table-name Music
```

Query a Local Secondary Index

You can use the `query` operation on a Local Secondary Index in much the same way that you `query` a table. You must specify the index name, the query criteria for the index sort key, and the attributes that you want to return. In this example, the index is `AlbumTitleIndex` and the index sort key is `AlbumTitle`.

The only attributes returned are those that have been projected into the index. You could modify this query to select non-key attributes too, but this would require table fetch activity that is relatively expensive. For more information about table fetches, see Attribute projections.

```
aws dynamodb query \
    --table-name Music \
    --index-name AlbumTitleIndex \
    --key-condition-expression "Artist = :v_artist and AlbumTitle = :v_title" \
    --expression-attribute-values  '{":v_artist":{"S":"Acme Band"},":v_title":
{"S":"Songs About Life"} }'
```

Managing complex workflows with DynamoDB transactions

Amazon DynamoDB transactions simplify the developer experience of making coordinated, all-or-nothing changes to multiple items both within and across tables. Transactions provide atomicity, consistency, isolation, and durability (ACID) in DynamoDB, helping you to maintain data correctness in your applications.

You can use the DynamoDB transactional read and write APIs to manage complex business workflows that require adding, updating, or deleting multiple items as a single, all-or-nothing operation. For example, a video game developer can ensure that players' profiles are updated correctly when they exchange items in a game or make in-game purchases.

With the transaction write API, you can group multiple `Put`, `Update`, `Delete`, and `ConditionCheck` actions. You can then submit the actions as a single `TransactWriteItems` operation that either succeeds or fails as a unit. The same is true for multiple `Get` actions, which you can group and submit as a single `TransactGetItems` operation.

There is no additional cost to enable transactions for your DynamoDB tables. You pay only for the reads or writes that are part of your transaction. DynamoDB performs two underlying reads or writes of every item in the transaction: one to prepare the transaction and one to commit the transaction. These two underlying read/write operations are visible in your Amazon CloudWatch metrics.

To get started with DynamoDB transactions, download the latest AWS SDK or the AWS Command Line Interface (AWS CLI). Then follow the DynamoDB transactions example.

The following sections provide a detailed overview of the transaction APIs and how you can use them in DynamoDB.

Topics

- Amazon DynamoDB Transactions: How it works
- Using IAM with DynamoDB transactions
- DynamoDB transactions example

Amazon DynamoDB Transactions: How it works

With Amazon DynamoDB transactions, you can group multiple actions together and submit them as a single all-or-nothing `TransactWriteItems` or `TransactGetItems` operation. The following sections describe API operations, capacity management, best practices, and other details about using transactional operations in DynamoDB.

Topics

- TransactWriteItems API
- TransactGetItems API
- Isolation levels for DynamoDB transactions
- Transaction conflict handling in DynamoDB
- Using transactional APIs in DynamoDB Accelerator (DAX)
- Capacity management for transactions
- Best practices for transactions
- Using transactional APIs with global tables
- DynamoDB Transactions vs. the AWSLabs transactions client library

TransactWriteItems API

`TransactWriteItems` is a synchronous and idempotent write operation that groups up to 100 write actions in a single all-or-nothing operation. These actions can target up to 100 distinct items in one or more DynamoDB tables within the same AWS account and in the same Region. The aggregate size of the items in the transaction cannot exceed 4 MB. The actions are completed atomically so that either all of them succeed or none of them succeeds.

Note

- A `TransactWriteItems` operation differs from a `BatchWriteItem` operation in that all the actions it contains must be completed successfully, or no changes are made at all. With a `BatchWriteItem` operation, it is possible that only some of the actions in the batch succeed while the others do not.
- Transactions cannot be performed using indexes.

You can't target the same item with multiple operations within the same transaction. For example, you can't perform a `ConditionCheck` and also an `Update` action on the same item in the same transaction.

You can add the following types of actions to a transaction:

- `Put` — Initiates a `PutItem` operation to create a new item or replace an old item with a new item, conditionally or without specifying any condition.
- `Update` — Initiates an `UpdateItem` operation to edit an existing item's attributes or add a new item to the table if it does not already exist. Use this action to add, delete, or update attributes on an existing item conditionally or without a condition.
- `Delete` — Initiates a `DeleteItem` operation to delete a single item in a table identified by its primary key.
- `ConditionCheck` — Checks that an item exists or checks the condition of specific attributes of the item.

When a transaction completes in DynamoDB, its changes start propagating to global secondary indexes (GSIs), streams, and backups. This propagation occurs gradually: stream records from the same transaction might appear at different times and could be interleaved with records from other transactions. Stream consumers shouldn't assume transaction atomicity or ordering guarantees.

To ensure an atomic snapshot of items modified in a transaction, use the TransactGetItems operation to read all relevant items together. This operation provides a consistent view of the data, ensuring you see either all changes from a completed transaction or none at all.

Because propagation isn't immediate, if a table is restored from backup (RestoreTableFromBackup) or exported to a point in time (ExportTableToPointInTime) mid-propagation,, it can contain only some of the changes made during a recent transaction.

Idempotency

You can optionally include a client token when you make a `TransactWriteItems` call to ensure that the request is *idempotent*. Making your transactions idempotent helps prevent application errors if the same operation is submitted multiple times due to a connection time-out or other connectivity issue.

If the original `TransactWriteItems` call was successful, then subsequent `TransactWriteItems` calls with the same client token return successfully without making any changes. If the `ReturnConsumedCapacity` parameter is set, the initial `TransactWriteItems` call returns the number of write capacity units consumed in making the changes. Subsequent `TransactWriteItems` calls with the same client token return the number of read capacity units consumed in reading the item.

Important points about idempotency

- A client token is valid for 10 minutes after the request that uses it finishes. After 10 minutes, any request that uses the same client token is treated as a new request. You should not reuse the same client token for the same request after 10 minutes.
- If you repeat a request with the same client token within the 10-minute idempotency window but change some other request parameter, DynamoDB returns an `IdempotentParameterMismatch` exception.

Error handling for writing

Write transactions don't succeed under the following circumstances:

- When a condition in one of the condition expressions is not met.
- When a transaction validation error occurs because more than one action in the same `TransactWriteItems` operation targets the same item.
- When a `TransactWriteItems` request conflicts with an ongoing `TransactWriteItems` operation on one or more items in the `TransactWriteItems` request. In this case, the request fails with a `TransactionCanceledException`.
- When there is insufficient provisioned capacity for the transaction to be completed.
- When an item size becomes too large (larger than 400 KB), or a local secondary index (LSI) becomes too large, or a similar validation error occurs because of changes made by the transaction.

- When there is a user error, such as an invalid data format.

For more information about how conflicts with `TransactWriteItems` operations are handled, see Transaction conflict handling in DynamoDB.

TransactGetItems API

`TransactGetItems` is a synchronous read operation that groups up to 100 `Get` actions together. These actions can target up to 100 distinct items in one or more DynamoDB tables within the same AWS account and Region. The aggregate size of the items in the transaction can't exceed 4 MB.

The `Get` actions are performed atomically so that either all of them succeed or all of them fail:

- `Get` — Initiates a `GetItem` operation to retrieve a set of attributes for the item with the given primary key. If no matching item is found, `Get` does not return any data.

Error handling for reading

Read transactions don't succeed under the following circumstances:

- When a `TransactGetItems` request conflicts with an ongoing `TransactWriteItems` operation on one or more items in the `TransactGetItems` request. In this case, the request fails with a `TransactionCanceledException`.
- When there is insufficient provisioned capacity for the transaction to be completed.
- When there is a user error, such as an invalid data format.

For more information about how conflicts with `TransactGetItems` operations are handled, see Transaction conflict handling in DynamoDB.

Isolation levels for DynamoDB transactions

The isolation levels of transactional operations (`TransactWriteItems` or `TransactGetItems`) and other operations are as follows.

SERIALIZABLE

Serializable isolation ensures that the results of multiple concurrent operations are the same as if no operation begins until the previous one has finished.

There is serializable isolation between the following types of operation:

- Between any transactional operation and any standard write operation (`PutItem`, `UpdateItem`, or `DeleteItem`).
- Between any transactional operation and any standard read operation (`GetItem`).
- Between a `TransactWriteItems` operation and a `TransactGetItems` operation.

Although there is serializable isolation between transactional operations, and each individual standard write in a `BatchWriteItem` operation, there is no serializable isolation between the transaction and the `BatchWriteItem` operation as a unit.

Similarly, the isolation level between a transactional operation and individual `GetItems` in a `BatchGetItem` operation is serializable. But the isolation level between the transaction and the `BatchGetItem` operation as a unit is *read-committed*.

A single `GetItem` request is serializable with respect to a `TransactWriteItems` request in one of two ways, either before or after the `TransactWriteItems` request. Multiple `GetItem` requests, against keys in a concurrent `TransactWriteItems` requests can be run in any order, and therefore the results are *read-committed*.

For example, if `GetItem` requests for item A and item B are run concurrently with a `TransactWriteItems` request that modifies both item A and item B, there are four possibilities:

- Both `GetItem` requests are run before the `TransactWriteItems` request.
- Both `GetItem` requests are run after the `TransactWriteItems` request.
- `GetItem` request for item A is run before the `TransactWriteItems` request. For item B the `GetItem` is run after `TransactWriteItems`.
- `GetItem` request for item B is run before the `TransactWriteItems` request. For item A the `GetItem` is run after `TransactWriteItems`.

You should use `TransactGetItems` if you prefer serializable isolation level for multiple `GetItem` requests.

If a non-transactional read is made on multiple items that were part of the same transaction write request in-flight, it's possible that you'll be able to read the new state of some of the items and the old state of the other items. You'll be able to read the new state of all items that were part of the

transaction write request only when a successful response is received for the transactional write, indicating that the transaction has been completed.

Once the transaction is successfully completed and a response is received, subsequent *eventually consistent* read operations may still return the old state for a short period due to DynamoDB's eventual consistency model. To guarantee reading the most up-to-date data immediately after a transaction, you should use *strongly consistent* reads by setting `ConsistentRead` to true.

READ-COMMITTED

Read-committed isolation ensures that read operations always return committed values for an item - the read will never present a view to the item representing a state from a transactional write which did not ultimately succeed. Read-committed isolation does not prevent modifications of the item immediately after the read operation.

The isolation level is read-committed between any transactional operation and any read operation that involves multiple standard reads (`BatchGetItem`, `Query`, or `Scan`). If a transactional write updates an item in the middle of a `BatchGetItem`, `Query`, or `Scan` operation, the subsequent part of the read operation returns the newly committed value (with `ConsistentRead`) or possibly a prior committed value (eventually consistent reads).

Operation summary

To summarize, the following table shows the isolation levels between a transaction operation (`TransactWriteItems` or `TransactGetItems`) and other operations.

Operation	Isolation Level
`DeleteItem`	*Serializable*
`PutItem`	*Serializable*
`UpdateItem`	*Serializable*
`GetItem`	*Serializable*
`BatchGetItem`	*Read-committed**
`BatchWriteItem`	*NOT Serializable**
`Query`	*Read-committed*

Operation	Isolation Level
Scan	*Read-committed*
Other transactional operation	*Serializable*

Levels marked with an asterisk (*) apply to the operation as a unit. However, individual actions within those operations have a *serializable* isolation level.

Transaction conflict handling in DynamoDB

A transactional conflict can occur during concurrent item-level requests on an item within a transaction. Transaction conflicts can occur in the following scenarios:

- A `PutItem`, `UpdateItem`, or `DeleteItem` request for an item conflicts with an ongoing `TransactWriteItems` request that includes the same item.
- An item within a `TransactWriteItems` request is part of another ongoing `TransactWriteItems` request.
- An item within a `TransactGetItems` request is part of an ongoing `TransactWriteItems`, `BatchWriteItem`, `PutItem`, `UpdateItem`, or `DeleteItem` request.

Note

- When a `PutItem`, `UpdateItem`, or `DeleteItem` request is rejected, the request fails with a `TransactionConflictException`.
- If any item-level request within `TransactWriteItems` or `TransactGetItems` is rejected, the request fails with a `TransactionCanceledException`. If that request fails, AWS SDKs do not retry the request.

  If you are using the AWS SDK for Java, the exception contains the list of CancellationReasons, ordered according to the list of items in the `TransactItems` request parameter. For other languages, a string representation of the list is included in the exception's error message.
- If an ongoing `TransactWriteItems` or `TransactGetItems` operation conflicts with a concurrent `GetItem` request, both operations can succeed.

The TransactionConflict CloudWatch metric is incremented for each failed item-level request.

Using transactional APIs in DynamoDB Accelerator (DAX)

`TransactWriteItems` and `TransactGetItems` are both supported in DynamoDB Accelerator (DAX) with the same isolation levels as in DynamoDB.

`TransactWriteItems` writes through DAX. DAX passes a `TransactWriteItems` call to DynamoDB and returns the response. To populate the cache after the write, DAX calls `TransactGetItems` in the background for each item in the `TransactWriteItems` operation, which consumes additional read capacity units. (For more information, see Capacity management for transactions.) This functionality enables you to keep your application logic simple and use DAX for both transactional operations and nontransactional ones.

`TransactGetItems` calls are passed through DAX without the items being cached locally. This is the same behavior as for strongly consistent read APIs in DAX.

Capacity management for transactions

There is no additional cost to enable transactions for your DynamoDB tables. You pay only for the reads or writes that are part of your transaction. DynamoDB performs two underlying reads or writes of every item in the transaction: one to prepare the transaction and one to commit the transaction. The two underlying read/write operations are visible in your Amazon CloudWatch metrics.

Plan for the additional reads and writes that are required by transactional APIs when you are provisioning capacity to your tables. For example, suppose that your application runs one transaction per second, and each transaction writes three 500-byte items in your table. Each item requires two write capacity units (WCUs): one to prepare the transaction and one to commit the transaction. Therefore, you would need to provision six WCUs to the table.

If you were using DynamoDB Accelerator (DAX) in the previous example, you would also use two read capacity units (RCUs) for each item in the `TransactWriteItems` call. So you would need to provision six additional RCUs to the table.

Similarly, if your application runs one read transaction per second, and each transaction reads three 500-byte items in your table, you would need to provision six read capacity units (RCUs) to the table. Reading each item requires two RCUs: one to prepare the transaction and one to commit the transaction.

Also, default SDK behavior is to retry transactions in case of a `TransactionInProgressException` exception. Plan for the additional read-capacity units (RCUs) that these retries consume. The same is true if you are retrying transactions in your own code using a `ClientRequestToken`.

Best practices for transactions

Consider the following recommended practices when using DynamoDB transactions.

- Enable automatic scaling on your tables, or ensure that you have provisioned enough throughput capacity to perform the two read or write operations for every item in your transaction.
- If you are not using an AWS provided SDK, include a `ClientRequestToken` attribute when you make a `TransactWriteItems` call to ensure that the request is idempotent.
- Don't group operations together in a transaction if it's not necessary. For example, if a single transaction with 10 operations can be broken up into multiple transactions without compromising the application correctness, we recommend splitting up the transaction. Simpler transactions improve throughput and are more likely to succeed.
- Multiple transactions updating the same items simultaneously can cause conflicts that cancel the transactions. We recommend following DynamoDB best practices for data modeling to minimize such conflicts.
- If a set of attributes is often updated across multiple items as part of a single transaction, consider grouping the attributes into a single item to reduce the scope of the transaction.
- Avoid using transactions for ingesting data in bulk. For bulk writes, it is better to use `BatchWriteItem`.

Using transactional APIs with global tables

Transactional operations provide atomicity, consistency, isolation, and durability (ACID) guarantees only within the AWS Region where the write API was invoked. Transactions aren't supported across Regions in global tables. For example, suppose that you have a global table with replicas in the US East (Ohio) and US West (Oregon) Regions and you perform a `TransactWriteItems` operation in the US East (N. Virginia) Region. You may observe partially completed transactions in the US West (Oregon) Region as changes are replicated. Changes are replicated to other Regions only after they've been committed in the source Region.

DynamoDB Transactions vs. the AWSLabs transactions client library

DynamoDB transactions provide a more cost-effective, robust, and performant replacement for the AWSLabs transactions client library. We suggest that you update your applications to use the native, server-side transaction APIs.

Using IAM with DynamoDB transactions

You can use AWS Identity and Access Management (IAM) to restrict the actions that transactional operations can perform in Amazon DynamoDB. For more information about using IAM policies in DynamoDB, see Identity-based policies for DynamoDB.

Permissions for `Put`, `Update`, `Delete`, and `Get` actions are governed by the permissions used for the underlying `PutItem`, `UpdateItem`, `DeleteItem`, and `GetItem` operations. For the `ConditionCheck` action, you can use the `dynamodb:ConditionCheckItem` permission in IAM policies.

The following are examples of IAM policies that you can use to configure the DynamoDB transactions.

Example 1: Allow transactional operations

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "dynamodb:ConditionCheckItem",
                "dynamodb:PutItem",
                "dynamodb:UpdateItem",
                "dynamodb:DeleteItem",
                "dynamodb:GetItem"
            ],
            "Resource": [
                "arn:aws:dynamodb:*:*:table/table04"
            ]
        }
    ]
```

```
}
```

Example 2: Allow only transactional operations

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "dynamodb:ConditionCheckItem",
                "dynamodb:PutItem",
                "dynamodb:UpdateItem",
                "dynamodb:DeleteItem",
                "dynamodb:GetItem"
            ],
            "Resource": [
                "arn:aws:dynamodb:*:*:table/table04"
            ],
            "Condition": {
                "ForAnyValue:StringEquals": {
                    "dynamodb:EnclosingOperation": [
                        "TransactWriteItems",
                        "TransactGetItems"
                    ]
                }
            }
        }
    ]
}
```

Example 3: Allow nontransactional reads and writes, and block transactional reads and writes

JSON

```
{
```

```
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Deny",
            "Action": [
                "dynamodb:ConditionCheckItem",
                "dynamodb:PutItem",
                "dynamodb:UpdateItem",
                "dynamodb:DeleteItem",
                "dynamodb:GetItem"
            ],
            "Resource": [
                "arn:aws:dynamodb:*:*:table/table04"
            ],
            "Condition": {
                "ForAnyValue:StringEquals": {
                    "dynamodb:EnclosingOperation": [
                        "TransactWriteItems",
                        "TransactGetItems"
                    ]
                }
            }
        },
        {
            "Effect": "Allow",
             "Action": [
                 "dynamodb:PutItem",
                 "dynamodb:DeleteItem",
                 "dynamodb:GetItem",
                 "dynamodb:UpdateItem"
             ],
             "Resource": [
                 "arn:aws:dynamodb:*:*:table/table04"
             ]
         }
    ]
}
```

Example 4: Prevent information from being returned on a ConditionCheck failure

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "dynamodb:ConditionCheckItem",
                "dynamodb:PutItem",
                "dynamodb:UpdateItem",
                "dynamodb:DeleteItem",
                "dynamodb:GetItem"
            ],
            "Resource": "arn:aws:dynamodb:*:*:table/table01",
            "Condition": {
                "StringEqualsIfExists": {
                    "dynamodb:ReturnValues": "NONE"
                }
            }
        }
    ]
}
```

DynamoDB transactions example

As an example of a situation in which Amazon DynamoDB transactions can be useful, consider this sample Java application for an online marketplace.

The application has three DynamoDB tables in the backend:

- `Customers` — This table stores details about the marketplace customers. Its primary key is a `CustomerId` unique identifier.
- `ProductCatalog` — This table stores details such as price and availability about the products for sale in the marketplace. Its primary key is a `ProductId` unique identifier.
- `Orders` — This table stores details about orders from the marketplace. Its primary key is an `OrderId` unique identifier.

Making an order

The following code snippets illustrate how to use DynamoDB transactions to coordinate the multiple steps that are required to create and process an order. Using a single all-or-nothing operation ensures that if any part of the transaction fails, no actions in the transaction are run and no changes are made.

In this example, you set up an order from a customer whose `customerId` is `09e8e9c8-ec48`. You then run it as a single transaction using the following simple order-processing workflow:

1. Determine that the customer ID is valid.
2. Make sure that the product is `IN_STOCK`, and update the product status to `SOLD`.
3. Make sure that the order does not already exist, and create the order.

Validate the customer

First, define an action to verify that a customer with `customerId` equal to `09e8e9c8-ec48` exists in the customer table.

```
final String CUSTOMER_TABLE_NAME = "Customers";
final String CUSTOMER_PARTITION_KEY = "CustomerId";
final String customerId = "09e8e9c8-ec48";
final HashMap<String, AttributeValue> customerItemKey = new HashMap<>();
customerItemKey.put(CUSTOMER_PARTITION_KEY, new AttributeValue(customerId));

ConditionCheck checkCustomerValid = new ConditionCheck()
    .withTableName(CUSTOMER_TABLE_NAME)
    .withKey(customerItemKey)
    .withConditionExpression("attribute_exists(" + CUSTOMER_PARTITION_KEY + ")");
```

Update the product status

Next, define an action to update the product status to `SOLD` if the condition that the product status is currently set to `IN_STOCK` is `true`. Setting the `ReturnValuesOnConditionCheckFailure` parameter returns the item if the item's product status attribute was not equal to `IN_STOCK`.

```
final String PRODUCT_TABLE_NAME = "ProductCatalog";
final String PRODUCT_PARTITION_KEY = "ProductId";
HashMap<String, AttributeValue> productItemKey = new HashMap<>();
productItemKey.put(PRODUCT_PARTITION_KEY, new AttributeValue(productKey));
```

```
Map<String, AttributeValue> expressionAttributeValues = new HashMap<>();
expressionAttributeValues.put(":new_status", new AttributeValue("SOLD"));
expressionAttributeValues.put(":expected_status", new AttributeValue("IN_STOCK"));

Update markItemSold = new Update()
    .withTableName(PRODUCT_TABLE_NAME)
    .withKey(productItemKey)
    .withUpdateExpression("SET ProductStatus = :new_status")
    .withExpressionAttributeValues(expressionAttributeValues)
    .withConditionExpression("ProductStatus = :expected_status")

 .withReturnValuesOnConditionCheckFailure(ReturnValuesOnConditionCheckFailure.ALL_OLD);
```

Create the order

Lastly, create the order as long as an order with that `OrderId` does not already exist.

```
final String ORDER_PARTITION_KEY = "OrderId";
final String ORDER_TABLE_NAME = "Orders";

HashMap<String, AttributeValue> orderItem = new HashMap<>();
orderItem.put(ORDER_PARTITION_KEY, new AttributeValue(orderId));
orderItem.put(PRODUCT_PARTITION_KEY, new AttributeValue(productKey));
orderItem.put(CUSTOMER_PARTITION_KEY, new AttributeValue(customerId));
orderItem.put("OrderStatus", new AttributeValue("CONFIRMED"));
orderItem.put("OrderTotal", new AttributeValue("100"));

Put createOrder = new Put()
    .withTableName(ORDER_TABLE_NAME)
    .withItem(orderItem)

 .withReturnValuesOnConditionCheckFailure(ReturnValuesOnConditionCheckFailure.ALL_OLD)
    .withConditionExpression("attribute_not_exists(" + ORDER_PARTITION_KEY + ")");
```

Run the transaction

The following example illustrates how to run the actions defined previously as a single all-or-nothing operation.

```
    Collection<TransactWriteItem> actions = Arrays.asList(
        new TransactWriteItem().withConditionCheck(checkCustomerValid),
        new TransactWriteItem().withUpdate(markItemSold),
```

```
        new TransactWriteItem().withPut(createOrder));

    TransactWriteItemsRequest placeOrderTransaction = new TransactWriteItemsRequest()
        .withTransactItems(actions)
        .withReturnConsumedCapacity(ReturnConsumedCapacity.TOTAL);

    // Run the transaction and process the result.
    try {
        client.transactWriteItems(placeOrderTransaction);
        System.out.println("Transaction Successful");

    } catch (ResourceNotFoundException rnf) {
        System.err.println("One of the table involved in the transaction is not found"
 + rnf.getMessage());
    } catch (InternalServerErrorException ise) {
        System.err.println("Internal Server Error" + ise.getMessage());
    } catch (TransactionCanceledException tce) {
        System.out.println("Transaction Canceled " + tce.getMessage());
    }
```

Reading the order details

The following example shows how to read the completed order transactionally across the `Orders` and `ProductCatalog` tables.

```
HashMap<String, AttributeValue> productItemKey = new HashMap<>();
productItemKey.put(PRODUCT_PARTITION_KEY, new AttributeValue(productKey));

HashMap<String, AttributeValue> orderKey = new HashMap<>();
orderKey.put(ORDER_PARTITION_KEY, new AttributeValue(orderId));

Get readProductSold = new Get()
    .withTableName(PRODUCT_TABLE_NAME)
    .withKey(productItemKey);
Get readCreatedOrder = new Get()
    .withTableName(ORDER_TABLE_NAME)
    .withKey(orderKey);

Collection<TransactGetItem> getActions = Arrays.asList(
    new TransactGetItem().withGet(readProductSold),
    new TransactGetItem().withGet(readCreatedOrder));

TransactGetItemsRequest readCompletedOrder = new TransactGetItemsRequest()
```

```
    .withTransactItems(getActions)
    .withReturnConsumedCapacity(ReturnConsumedCapacity.TOTAL);

// Run the transaction and process the result.
try {
    TransactGetItemsResult result = client.transactGetItems(readCompletedOrder);
    System.out.println(result.getResponses());
} catch (ResourceNotFoundException rnf) {
    System.err.println("One of the table involved in the transaction is not found" +
 rnf.getMessage());
} catch (InternalServerErrorException ise) {
    System.err.println("Internal Server Error" + ise.getMessage());
} catch (TransactionCanceledException tce) {
    System.err.println("Transaction Canceled" + tce.getMessage());
}
```

Change data capture with Amazon DynamoDB

Many applications benefit from capturing changes to items stored in a DynamoDB table, at the point in time when such changes occur. The following are some example use cases:

- A popular mobile app modifies data in a DynamoDB table, at the rate of thousands of updates per second. Another application captures and stores data about these updates, providing near-real-time usage metrics for the mobile app.
- A financial application modifies stock market data in a DynamoDB table. Different applications running in parallel track these changes in real time, compute value-at-risk, and automatically rebalance portfolios based on stock price movements.
- Sensors in transportation vehicles and industrial equipment send data to a DynamoDB table. Different applications monitor performance and send messaging alerts when a problem is detected, predict any potential defects by applying machine learning algorithms, and compress and archive data to Amazon Simple Storage Service (Amazon S3).
- An application automatically sends notifications to the mobile devices of all friends in a group as soon as one friend uploads a new picture.
- A new customer adds data to a DynamoDB table. This event invokes another application that sends a welcome email to the new customer.

DynamoDB supports streaming of item-level change data capture records in near-real time. You can build applications that consume these streams and take action based on the contents.

> **Note**
>
> Adding tags to DynamoDB Streams and using attribute-based access control (ABAC) with DynamoDB Streams aren't supported.

The following video will give you an introductory look at the change data capture concept.

Table capacity modes

Topics

- Streaming options for change data capture
- Using Kinesis Data Streams to capture changes to DynamoDB
- Change data capture for DynamoDB Streams

Streaming options for change data capture

DynamoDB offers two streaming models for change data capture: Kinesis Data Streams for DynamoDB and DynamoDB Streams.

To help you choose the right solution for your application, the following table summarizes the features of each streaming model.

Properties	Kinesis Data Streams for DynamoDB	DynamoDB Streams
Data retention	Up to 1 year.	24 hours.
Kinesis Client Library (KCL) support	Supports KCL versions 1.X, 2.X, and 3.X.	Supports KCL versions 1.X and 2.X.
Number of consumers	Up to 5 simultaneous consumers per shard, or up to 20 simultaneous consumers per shard with enhanced fan-out.	Up to 2 simultaneous consumers per shard.

Properties	Kinesis Data Streams for DynamoDB	DynamoDB Streams
Throughput quotas	Unlimited.	Subject to throughput quotas by DynamoDB table and AWS Region.
Record delivery model	Pull model over HTTP using GetRecords and with enhanced fan-out, Kinesis Data Streams pushes the records over HTTP/2 by using SubscribeToShard.	Pull model over HTTP using GetRecords.
Ordering of records	The timestamp attribute on each stream record can be used to identify the actual order in which changes occurred in the DynamoDB table.	For each item that is modified in a DynamoDB table, the stream records appear in the same sequence as the actual modifications to the item.
Duplicate records	Duplicate records might occasionally appear in the stream.	No duplicate records appear in the stream.
Stream processing options	Process stream records using AWS Lambda, Amazon Managed Service for Apache Flink, Kinesis data firehose , or AWS Glue streaming ETL.	Process stream records using AWS Lambda or DynamoDB Streams Kinesis adapter.
Durability level	Availability zones to provide automatic failover without interruption.	Availability zones to provide automatic failover without interruption.

You can enable both streaming models on the same DynamoDB table.

The following video talks more about the differences between the two options.

DynamoDB Streams vs Kinesis Data Streams

Using Kinesis Data Streams to capture changes to DynamoDB

You can use Amazon Kinesis Data Streams to capture changes to Amazon DynamoDB.

Kinesis Data Streams captures item-level modifications in any DynamoDB table and replicates them to a Kinesis data stream. Your applications can access this stream and view item-level changes in near-real time. You can continuously capture and store terabytes of data per hour. You can take advantage of longer data retention time—and with enhanced fan-out capability, you can simultaneously reach two or more downstream applications. Other benefits include additional audit and security transparency.

Kinesis Data Streams also gives you access to Amazon Data Firehose and Amazon Managed Service for Apache Flink. These services can help you build applications that power real-time dashboards, generate alerts, implement dynamic pricing and advertising, and implement sophisticated data analytics and machine learning algorithms.

> **Note**
>
> Using Kinesis data streams for DynamoDB is subject to both Kinesis Data Streams pricing for the data stream and DynamoDB pricing for the source table.

To enable Kinesis streaming on a DynamoDB table using the console, AWS CLI, or Java SDK, see Getting started with Kinesis Data Streams for Amazon DynamoDB.

Topics

- How Kinesis Data Streams works with DynamoDB
- Getting started with Kinesis Data Streams for Amazon DynamoDB
- Using shards and metrics with DynamoDB Streams and Kinesis Data Streams
- Using IAM policies for Amazon Kinesis Data Streams and Amazon DynamoDB

How Kinesis Data Streams works with DynamoDB

When a Kinesis data stream is enabled for a DynamoDB table, the table sends out a data record that captures any changes to that table's data. This data record includes:

- The specific time any item was recently created, updated, or deleted
- That item's primary key
- A snapshot of the record before the modification
- A snapshot of the record after the modification

These data records are captured and published in near-real time. After they are written to the Kinesis data stream, they can be read just like any other record. You can use the Kinesis Client Library, use AWS Lambda, call the Kinesis Data Streams API, and use other connected services. For more information, see Reading Data from Amazon Kinesis Data Streams in the Amazon Kinesis Data Streams Developer Guide.

These changes to data are also captured asynchronously. Kinesis has no performance impact on a table that it's streaming from. The stream records stored in your Kinesis data stream are also encrypted at rest. For more information, see Data Protection in Amazon Kinesis Data Streams.

The Kinesis data stream records might appear in a different order than when the item changes occurred. The same item notifications might also appear more than once in the stream. You can check the `ApproximateCreationDateTime` attribute to identify the order that the item modifications occurred in, and to identify duplicate records.

When you enable a Kinesis data stream as a streaming destination of a DynamoDB table, you can configure the precision of `ApproximateCreationDateTime` values in either milliseconds or microseconds. By default, `ApproximateCreationDateTime` indicates the time of the change in milliseconds. Additionally, you can change this value on an active streaming destination. After such an update, stream records written to Kinesis will have `ApproximateCreationDateTime` values of the desired precision.

Binary values written to DynamoDB must be encoded in base64-encoded format . However, when data records are written to a Kinesis data stream, these encoded binary values are encoded with base64-encoding a second time. When reading these records from a Kinesis data stream, in order to retrieve the raw binary values, applications must decode these values twice.

DynamoDB charges for using Kinesis Data Streams in change data capture units. 1 KB of change per single item counts as one change data capture unit. The KB of change in each item is calculated by the larger of the "before" and "after" images of the item written to the stream, using the same logic as capacity unit consumption for write operations. Similar to how DynamoDB on-demand mode works, you don't need to provision capacity throughput for change data capture units.

Turning on a Kinesis data stream for your DynamoDB table

You can enable or disable streaming to Kinesis from your existing DynamoDB table by using the AWS Management Console, the AWS SDK, or the AWS Command Line Interface (AWS CLI).

- You can only stream data from DynamoDB to Kinesis Data Streams in the same AWS account and AWS Region as your table.
- You can only stream data from a DynamoDB table to one Kinesis data stream.

Making changes to a Kinesis Data Streams destination on your DynamoDB table

By default, all Kinesis data stream records include an `ApproximateCreationDateTime` attribute. This attribute represents a timestamp in milliseconds of the approximate time when each record was created. You can change the precision of these values by using the https://console.aws.amazon.com/kinesis, the SDK or the AWS CLI

Getting started with Kinesis Data Streams for Amazon DynamoDB

This section describes how to use Kinesis Data Streams for Amazon DynamoDB tables with the Amazon DynamoDB console, the AWS Command Line Interface (AWS CLI), and the API.

Creating an active Amazon Kinesis data stream

All of these examples use the `Music` DynamoDB table that was created as part of the Getting started with DynamoDB tutorial.

To learn more about how to build consumers and connect your Kinesis data stream to other AWS services, see Reading data from Kinesis Data Streams in the *Amazon Kinesis Data Streams developer guide*.

> **Note**
>
> When you're first using KDS shards, we recommend setting your shards to scale up and down with usage patterns. After you have accumulated more data on usage patterns, you can adjust the shards in your stream to match.

Console

1. Sign in to the AWS Management Console and open the Kinesis console at https://console.aws.amazon.com/kinesis/.
2. Choose **Create data stream** and follow the instructions to create a stream called `samplestream`.
3. Open the DynamoDB console at https://console.aws.amazon.com/dynamodb/.
4. In the navigation pane on the left side of the console, choose **Tables**.
5. Choose the **Music** table.
6. Choose the **Exports and streams** tab.
7. (Optional) Under **Amazon Kinesis data stream details**, you can change the record timestamp precision from microsecond (default) to millisecond.
8. Choose **samplestream** from the dropdown list.
9. Choose the **Turn On** button.

AWS CLI

1. Create a Kinesis Data Streams named `samplestream` by using the create-stream command.

   ```
   aws kinesis create-stream --stream-name samplestream --shard-count 3
   ```

   See Shard management considerations for Kinesis Data Streams before setting the number of shards for the Kinesis data stream.
2. Check that the Kinesis stream is active and ready for use by using the describe-stream command.

   ```
   aws kinesis describe-stream --stream-name samplestream
   ```
3. Enable Kinesis streaming on the DynamoDB table by using the DynamoDB `enable-kinesis-streaming-destination` command. Replace the `stream-arn` value with the one that was returned by `describe-stream` in the previous step. Optionally, enable streaming with a more granular (microsecond) precision of timestamp values returned on each record.

   Enable streaming with microsecond timestamp precision:

```
aws dynamodb enable-kinesis-streaming-destination \
  --table-name Music \
  --stream-arn arn:aws:kinesis:us-west-2:12345678901:stream/samplestream
  --enable-kinesis-streaming-configuration
 ApproximateCreationDateTimePrecision=MICROSECOND
```

Or enable streaming with default timestamp precision (millisecond):

```
aws dynamodb enable-kinesis-streaming-destination \
  --table-name Music \
  --stream-arn arn:aws:kinesis:us-west-2:12345678901:stream/samplestream
```

4. Check if Kinesis streaming is active on the table by using the DynamoDB `describe-kinesis-streaming-destination` command.

```
aws dynamodb describe-kinesis-streaming-destination --table-name Music
```

5. Write data to the DynamoDB table by using the `put-item` command, as described in the DynamoDB Developer Guide.

```
aws dynamodb put-item \
    --table-name Music  \
    --item \
        '{"Artist": {"S": "No One You Know"}, "SongTitle": {"S": "Call Me
 Today"}, "AlbumTitle": {"S": "Somewhat Famous"}, "Awards": {"N": "1"}}'

aws dynamodb put-item \
    --table-name Music \
    --item \
        '{"Artist": {"S": "Acme Band"}, "SongTitle": {"S": "Happy Day"},
 "AlbumTitle": {"S": "Songs About Life"}, "Awards": {"N": "10"} }'
```

6. Use the Kinesis get-records CLI command to retrieve the Kinesis stream contents. Then use the following code snippet to deserialize the stream content.

```
/**
 * Takes as input a Record fetched from Kinesis and does arbitrary processing as
 an example.
 */
public void processRecord(Record kinesisRecord) throws IOException {
    ByteBuffer kdsRecordByteBuffer = kinesisRecord.getData();
```

```
    JsonNode rootNode = OBJECT_MAPPER.readTree(kdsRecordByteBuffer.array());
    JsonNode dynamoDBRecord = rootNode.get("dynamodb");
    JsonNode oldItemImage = dynamoDBRecord.get("OldImage");
    JsonNode newItemImage = dynamoDBRecord.get("NewImage");
    Instant recordTimestamp = fetchTimestamp(dynamoDBRecord);

    /**
     * Say for example our record contains a String attribute named "stringName"
 and we want to fetch the value
     * of this attribute from the new item image. The following code fetches
 this value.
     */
    JsonNode attributeNode = newItemImage.get("stringName");
    JsonNode attributeValueNode = attributeNode.get("S"); // Using DynamoDB "S"
 type attribute
    String attributeValue = attributeValueNode.textValue();
    System.out.println(attributeValue);
}

private Instant fetchTimestamp(JsonNode dynamoDBRecord) {
    JsonNode timestampJson = dynamoDBRecord.get("ApproximateCreationDateTime");
    JsonNode timestampPrecisionJson =
 dynamoDBRecord.get("ApproximateCreationDateTimePrecision");
    if (timestampPrecisionJson != null &&
 timestampPrecisionJson.equals("MICROSECOND")) {
        return Instant.EPOCH.plus(timestampJson.longValue(), ChronoUnit.MICROS);
    }
    return Instant.ofEpochMilli(timestampJson.longValue());
}
```

Java

1. Follow the instructions in the Kinesis Data Streams developer guide to create a Kinesis data stream named `samplestream` using Java.

   See Shard management considerations for Kinesis Data Streams before setting the number of shards for the Kinesis data stream.

2. Use the following code snippet to enable Kinesis streaming on the DynamoDB table. Optionally, enable streaming with a more granular (microsecond) precision of timestamp values returned on each record.

Enable streaming with microsecond timestamp precision:

```
EnableKinesisStreamingConfiguration enableKdsConfig =
 EnableKinesisStreamingConfiguration.builder()

 .approximateCreationDateTimePrecision(ApproximateCreationDateTimePrecision.MICROSECOND)
  .build();

EnableKinesisStreamingDestinationRequest enableKdsRequest =
 EnableKinesisStreamingDestinationRequest.builder()
  .tableName(tableName)
  .streamArn(kdsArn)
  .enableKinesisStreamingConfiguration(enableKdsConfig)
  .build();

EnableKinesisStreamingDestinationResponse enableKdsResponse =
 ddbClient.enableKinesisStreamingDestination(enableKdsRequest);
```

Or enable streaming with default timestamp precision (millisecond):

```
EnableKinesisStreamingDestinationRequest enableKdsRequest =
 EnableKinesisStreamingDestinationRequest.builder()
  .tableName(tableName)
  .streamArn(kdsArn)
  .build();

EnableKinesisStreamingDestinationResponse enableKdsResponse =
 ddbClient.enableKinesisStreamingDestination(enableKdsRequest);
```

3. Follow the instructions in the *Kinesis Data Streams developer guide* to read from the created data stream.
4. Use the following code snippet to deserialize the stream content

```
/**
 * Takes as input a Record fetched from Kinesis and does arbitrary processing as
 an example.
 */
public void processRecord(Record kinesisRecord) throws IOException {
    ByteBuffer kdsRecordByteBuffer = kinesisRecord.getData();
    JsonNode rootNode = OBJECT_MAPPER.readTree(kdsRecordByteBuffer.array());
    JsonNode dynamoDBRecord = rootNode.get("dynamodb");
```

```
    JsonNode oldItemImage = dynamoDBRecord.get("OldImage");
    JsonNode newItemImage = dynamoDBRecord.get("NewImage");
    Instant recordTimestamp = fetchTimestamp(dynamoDBRecord);

    /**
     * Say for example our record contains a String attribute named "stringName"
 and we wanted to fetch the value
     * of this attribute from the new item image, the below code would fetch
 this.
     */
    JsonNode attributeNode = newItemImage.get("stringName");
    JsonNode attributeValueNode = attributeNode.get("S"); // Using DynamoDB "S"
 type attribute
    String attributeValue = attributeValueNode.textValue();
    System.out.println(attributeValue);
}

private Instant fetchTimestamp(JsonNode dynamoDBRecord) {
    JsonNode timestampJson = dynamoDBRecord.get("ApproximateCreationDateTime");
    JsonNode timestampPrecisionJson =
 dynamoDBRecord.get("ApproximateCreationDateTimePrecision");
    if (timestampPrecisionJson != null &&
 timestampPrecisionJson.equals("MICROSECOND")) {
        return Instant.EPOCH.plus(timestampJson.longValue(), ChronoUnit.MICROS);
    }
    return Instant.ofEpochMilli(timestampJson.longValue());
}
```

Making changes to an active Amazon Kinesis data stream

This section describes how to make changes to an active Kinesis Data Streams for DynamoDB setup by using the console, AWS CLI and the API.

AWS Management Console

1. Open the DynamoDB console at https://console.aws.amazon.com/dynamodb/
2. Go to your table.
3. Choose **Exports and Streams**.

AWS CLI

1. Call `describe-kinesis-streaming-destination` to confirm that the stream is `ACTIVE`.
2. Call `UpdateKinesisStreamingDestination`, such as in this example:

```
aws dynamodb update-kinesis-streaming-destination --table-name
 enable_test_table --stream-arn arn:aws:kinesis:us-east-1:12345678901:stream/
enable_test_stream --update-kinesis-streaming-configuration
 ApproximateCreationDateTimePrecision=MICROSECOND
```

3. Call `describe-kinesis-streaming-destination` to confirm that the stream is `UPDATING`.
4. Call `describe-kinesis-streaming-destination` periodically until the streaming status is `ACTIVE` again. It typically takes up to 5 minutes for the timestamp precision updates to take effect. Once this status updates, that indicates that the update is complete and the new precision value will be applied on future records.
5. Write to the table using `putItem`.
6. Use the Kinesis `get-records` command to get the stream contents.
7. Confirm that the `ApproximateCreationDateTime` of the writes have the desired precision.

Java API

1. Provide a code snippet that constructs an `UpdateKinesisStreamingDestination` request and an `UpdateKinesisStreamingDestination` response.
2. Provide a code snippet that constructs a `DescribeKinesisStreamingDestination` request and a `DescribeKinesisStreamingDestination response`.
3. Call `describe-kinesis-streaming-destination` periodically until the streaming status is `ACTIVE` again, indicating that the update is complete and the new precision value will be applied on future records.
4. Perform writes to the table.
5. Read from the stream and deserialize the stream content.
6. Confirm that the `ApproximateCreationDateTime` of the writes have the desired precision.

Using shards and metrics with DynamoDB Streams and Kinesis Data Streams

Shard management considerations for Kinesis Data Streams

A Kinesis data stream counts its throughput in shards. In Amazon Kinesis Data streams, you can choose between an **on-demand** mode and a **provisioned** mode for your data streams.

We recommend using on-demand mode for your Kinesis Data Stream if your DynamoDB write workload is highly variable and unpredictable. With on-demand mode, there is no capacity planning required as Kinesis Data Streams automatically manages the shards in order to provide the necessary throughput.

For predictable workloads, you can use provisioned mode for your Kinesis Data Stream. With provisioned mode, you must specify the number of shards for the data stream to accommodate the change data capture records from DynamoDB. To determine the number of shards that the Kinesis data stream will need to support your DynamoDB table, you need the following input values:

- The average size of your DynamoDB table's record in bytes (`average_record_size_in_bytes`).
- The maximum number of write operations that your DynamoDB table will perform per second. This includes create, delete, and update operations performed by your applications, as well as automatically generated operations like Time to Live generated delete operations(`write_throughput`).
- The percentage of update and overwrite operations that you perform on your table, as compared to create or delete operations (`percentage_of_updates`). Keep in mind that update and overwrite operations replicate both the old and new images of the modified item to the stream. This generates twice the DynamoDB item size.

You can calculate the number of shards (`number_of_shards`) that your Kinesis data stream needs by using the input values in the following formula:

```
number_of_shards = ceiling( max( ((write_throughput * (4+percentage_of_updates) *
 average_record_size_in_bytes) / 1024 / 1024), (write_throughput/1000)), 1)
```

For example, you might have a maximum throughput of 1040 write operations per second (`write_throughput`) with an average record size of 800 bytes (`average_record_size_in_bytes`). If 25 percent of those write operations are update

operations (`percentage_of_updates`), then you will need two shards (`number_of_shards`) to accommodate your DynamoDB streaming throughput:

```
ceiling( max( ((1040 * (4+25/100) * 800)/ 1024 / 1024), (1040/1000)), 1).
```

Consider the following before using the formula to calculate the number of shards required with provisioned mode for Kinesis data streams:

- This formula helps estimate the number of shards that will be required to accommodate your DynamoDB change data records. It doesn't represent the total number of shards needed in your Kinesis data stream, such as the number of shards required to support additional Kinesis data stream consumers.
- You may still experience read and write throughput exceptions in the provisioned mode if you don't configure your data stream to handle your peak throughput. In this case, you must manually scale your data stream to accommodate your data traffic.
- This formula takes into consideration the additional bloat generated by DynamoDB before streaming the change logs data records to Kinesis Data Stream.

To learn more about capacity modes on Kinesis Data Stream see Choosing the Data Stream Capacity Mode. To learn more about pricing difference between different capacity modes, see Amazon Kinesis Data Streams pricing .

Monitoring change data capture with Kinesis Data Streams

DynamoDB provides several Amazon CloudWatch metrics to help you monitor the replication of change data capture to Kinesis. For a full list of CloudWatch metrics, see DynamoDB Metrics and dimensions.

To determine whether your stream has sufficient capacity, we recommend that you monitor the following items both during stream enabling and in production:

- `ThrottledPutRecordCount`: The number of records that were throttled by your Kinesis data stream because of insufficient Kinesis data stream capacity. You might experience some throttling during exceptional usage peaks, but the `ThrottledPutRecordCount` should remain as low as possible. DynamoDB retries sending throttled records to the Kinesis data stream, but this might result in higher replication latency.

If you experience excessive and regular throttling, you might need to increase the number of Kinesis stream shards proportionally to the observed write throughput of your table. To learn more about determining the size of a Kinesis data stream, see Determining the Initial Size of a Kinesis Data Stream.

- `AgeOfOldestUnreplicatedRecord`: The elapsed time since the oldest item-level change yet to replicate to the Kinesis data stream appeared in the DynamoDB table. Under normal operation, `AgeOfOldestUnreplicatedRecord` should be in the order of milliseconds. This number grows based on unsuccessful replication attempts when these are caused by customer-controlled configuration choices.

  If `AgeOfOldestUnreplicatedRecord` metric exceeds 168 hours, replication of item-level changes from the DynamoDB table to Kinesis data stream will be automatically disabled.

  Customer-controlled configuration examples that leads to unsuccessful replication attempts are an under-provisioned Kinesis data stream capacity that leads to excessive throttling, or a manual update to your Kinesis data stream's access policies that prevents DynamoDB from adding data to your data stream. To keep this metric as low as possible, you might need to ensure the right provisioning of your Kinesis data stream capacity, and make sure that DynamoDB's permissions are unchanged.
- `FailedToReplicateRecordCount`: The number of records that DynamoDB failed to replicate to your Kinesis data stream. Certain items larger than 34 KB might expand in size to change data records that are larger than the 1 MB item size limit of Kinesis Data Streams. This size expansion occurs when these larger than 34 KB items include a large number of Boolean or empty attribute values. Boolean and empty attribute values are stored as 1 byte in DynamoDB, but expand up to 5 bytes when they're serialized using standard JSON for Kinesis Data Streams replication. DynamoDB can't replicate such change records to your Kinesis data stream. DynamoDB skips these change data records, and automatically continues replicating subsequent records.

You can create Amazon CloudWatch alarms that send an Amazon Simple Notification Service (Amazon SNS) message for notification when any of the preceding metrics exceed a specific threshold.

Using IAM policies for Amazon Kinesis Data Streams and Amazon DynamoDB

The first time that you enable Amazon Kinesis Data Streams for Amazon DynamoDB, DynamoDB automatically creates an AWS Identity and Access Management (IAM) service-linked role for

you. This role, `AWSServiceRoleForDynamoDBKinesisDataStreamsReplication`, allows DynamoDB to manage the replication of item-level changes to Kinesis Data Streams on your behalf. Don't delete this service-linked role.

For more information about service-linked roles, see Using service-linked roles in the *IAM User Guide*.

> **Note**
>
> DynamoDB does not support tag-based conditions for IAM policies.

To enable Amazon Kinesis Data Streams for Amazon DynamoDB, you must have the following permissions on the table:

- `dynamodb:EnableKinesisStreamingDestination`
- `kinesis:ListStreams`
- `kinesis:PutRecords`
- `kinesis:DescribeStream`

To describe Amazon Kinesis Data Streams for Amazon DynamoDB for a given DynamoDB table, you must have the following permissions on the table.

- `dynamodb:DescribeKinesisStreamingDestination`
- `kinesis:DescribeStreamSummary`
- `kinesis:DescribeStream`

To disable Amazon Kinesis Data Streams for Amazon DynamoDB, you must have the following permissions on the table.

- `dynamodb:DisableKinesisStreamingDestination`

To update Amazon Kinesis Data Streams for Amazon DynamoDB, you must have the following permissions on the table.

- `dynamodb:UpdateKinesisStreamingDestination`

The following examples show how to use IAM policies to grant permissions for Amazon Kinesis Data Streams for Amazon DynamoDB.

Example: Enable Amazon Kinesis Data Streams for Amazon DynamoDB

The following IAM policy grants permissions to enable Amazon Kinesis Data Streams for Amazon DynamoDB for the `Music` table. It does not grant permissions to disable, update or describe Kinesis Data Streams for DynamoDB for the `Music` table.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "iam:CreateServiceLinkedRole",
            "Resource": "arn:aws:iam::*:role/aws-
service-role/kinesisreplication.dynamodb.amazonaws.com/
AWSServiceRoleForDynamoDBKinesisDataStreamsReplication",
            "Condition": {
                "StringLike": {
                    "iam:AWSServiceName":
 "kinesisreplication.dynamodb.amazonaws.com"
                }
            }
        },
        {
            "Effect": "Allow",
            "Action": [
                "dynamodb:EnableKinesisStreamingDestination"
            ],
            "Resource": "arn:aws:dynamodb:us-west-2:111122223333:table/Music"
        }
    ]
}
```

Example: Update Amazon Kinesis Data Streams for Amazon DynamoDB

The following IAM policy grants permissions to update Amazon Kinesis Data Streams for Amazon DynamoDB for the `Music` table. It does not grant permissions to enable, disable or describe Amazon Kinesis Data Streams for Amazon DynamoDB for the `Music` table.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "dynamodb:UpdateKinesisStreamingDestination"
            ],
            "Resource": "arn:aws:dynamodb:us-west-2:111122223333:table/Music"
        }
    ]
}
```

Example: Disable Amazon Kinesis Data Streams for Amazon DynamoDB

The following IAM policy grants permissions to disable Amazon Kinesis Data Streams for Amazon DynamoDB for the `Music` table. It does not grant permissions to enable, update or describe Amazon Kinesis Data Streams for Amazon DynamoDB for the `Music` table.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "dynamodb:DisableKinesisStreamingDestination"
            ],
            "Resource": "arn:aws:dynamodb:us-west-2:111122223333:table/Music"
        }
    ]
```

```
}
```

Example: Selectively apply permissions for Amazon Kinesis Data Streams for Amazon DynamoDB based on resource

The following IAM policy grants permissions to enable and describe Amazon Kinesis Data Streams for Amazon DynamoDB for the `Music` table, and denies permissions to disable Amazon Kinesis Data Streams for Amazon DynamoDB for the `Orders` table.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "dynamodb:EnableKinesisStreamingDestination",
                "dynamodb:DescribeKinesisStreamingDestination"
            ],
            "Resource": "arn:aws:dynamodb:us-west-2:111122223333:table/Music"
        },
        {
            "Effect": "Deny",
            "Action": [
                "dynamodb:DisableKinesisStreamingDestination"
            ],
            "Resource": "arn:aws:dynamodb:us-west-2:111122223333:table/Orders"
        }
    ]
}
```

Using service-linked roles for Kinesis Data Streams for DynamoDB

Amazon Kinesis Data Streams for Amazon DynamoDB uses AWS Identity and Access Management (IAM) service-linked roles. A service-linked role is a unique type of IAM role that is linked directly to Kinesis Data Streams for DynamoDB. Service-linked roles are predefined by Kinesis Data Streams for DynamoDB and include all the permissions that the service requires to call other AWS services on your behalf.

A service-linked role makes setting up Kinesis Data Streams for DynamoDB easier because you don't have to manually add the necessary permissions. Kinesis Data Streams for DynamoDB defines the permissions of its service-linked roles, and unless defined otherwise, only Kinesis Data Streams for DynamoDB can assume its roles. The defined permissions include the trust policy and the permissions policy, and that permissions policy cannot be attached to any other IAM entity.

For information about other services that support service-linked roles, see AWS Services That Work with IAM and look for the services that have **Yes** in the **Service-Linked Role** column. Choose a **Yes** with a link to view the service-linked role documentation for that service.

Service-linked role permissions for Kinesis Data Streams for DynamoDB

Kinesis Data Streams for DynamoDB uses the service-linked role named **AWSServiceRoleForDynamoDBKinesisDataStreamsReplication**. The purpose of the service-linked role is to allow Amazon DynamoDB to manage the replication of item-level changes to Kinesis Data Streams, on your behalf.

The `AWSServiceRoleForDynamoDBKinesisDataStreamsReplication` service-linked role trusts the following services to assume the role:

- `kinesisreplication.dynamodb.amazonaws.com`

The role permissions policy allows Kinesis Data Streams for DynamoDB to complete the following actions on the specified resources:

- Action: `Put records and describe` on `Kinesis stream`
- Action: `Generate data keys` on `AWS KMS` in order to put data on Kinesis streams that are encrypted using User-Generated AWS KMS keys.

For the exact contents of the policy document, see DynamoDBKinesisReplicationServiceRolePolicy.

You must configure permissions to allow an IAM entity (such as a user, group, or role) to create, edit, or delete a service-linked role. For more information, see Service-Linked Role Permissions in the *IAM User Guide*.

Creating a service-linked role for Kinesis Data Streams for DynamoDB

You don't need to manually create a service-linked role. When you enable Kinesis Data Streams for DynamoDB in the AWS Management Console, the AWS CLI, or the AWS API, Kinesis Data Streams for DynamoDB creates the service-linked role for you.

If you delete this service-linked role, and then need to create it again, you can use the same process to recreate the role in your account. When you enable Kinesis Data Streams for DynamoDB, Kinesis Data Streams for DynamoDB creates the service-linked role for you again.

Editing a service-linked role for Kinesis Data Streams for DynamoDB

Kinesis Data Streams for DynamoDB does not allow you to edit the `AWSServiceRoleForDynamoDBKinesisDataStreamsReplication` service-linked role. After you create a service-linked role, you cannot change the name of the role because various entities might reference the role. However, you can edit the description of the role using IAM. For more information, see Editing a Service-Linked Role in the *IAM User Guide*.

Deleting a service-linked role for Kinesis Data Streams for DynamoDB

You can also use the IAM console, the AWS CLI or the AWS API to manually delete the service-linked role. To do this, you must first manually clean up the resources for your service-linked role and then you can manually delete it.

> **Note**
>
> If the Kinesis Data Streams for DynamoDB service is using the role when you try to delete the resources, then the deletion might fail. If that happens, wait for a few minutes and try the operation again.

To manually delete the service-linked role using IAM

Use the IAM console, the AWS CLI, or the AWS API to delete the `AWSServiceRoleForDynamoDBKinesisDataStreamsReplication` service-linked role. For more information, see Deleting a Service-Linked Role in the *IAM User Guide*.

Change data capture for DynamoDB Streams

DynamoDB Streams captures a time-ordered sequence of item-level modifications in any DynamoDB table and stores this information in a log for up to 24 hours. Applications can access

this log and view the data items as they appeared before and after they were modified, in near-real time.

Encryption at rest encrypts the data in DynamoDB streams. For more information, see DynamoDB encryption at rest.

A *DynamoDB stream* is an ordered flow of information about changes to items in a DynamoDB table. When you enable a stream on a table, DynamoDB captures information about every modification to data items in the table.

Whenever an application creates, updates, or deletes items in the table, DynamoDB Streams writes a stream record with the primary key attributes of the items that were modified. A *stream record* contains information about a data modification to a single item in a DynamoDB table. You can configure the stream so that the stream records capture additional information, such as the "before" and "after" images of modified items.

DynamoDB Streams helps ensure the following:

- Each stream record appears exactly once in the stream.
- For each item that is modified in a DynamoDB table, the stream records appear in the same sequence as the actual modifications to the item.

DynamoDB Streams writes stream records in near-real time so that you can build applications that consume these streams and take action based on the contents.

Topics

- Endpoints for DynamoDB Streams
- Enabling a stream
- Reading and processing a stream
- DynamoDB Streams and Time to Live
- Using the DynamoDB Streams Kinesis adapter to process stream records
- DynamoDB Streams low-level API: Java example
- DynamoDB Streams and AWS Lambda triggers
- DynamoDB Streams and Apache Flink

Endpoints for DynamoDB Streams

AWS maintains separate endpoints for DynamoDB and DynamoDB Streams. To work with database tables and indexes, your application must access a DynamoDB endpoint. To read and process DynamoDB Streams records, your application must access a DynamoDB Streams endpoint in the same Region.

DynamoDB Streams offers two sets of endpoints. They are:

- **IPv4-only endpoints**: Endpoints with the `streams.dynamodb.<region>.amazonaws.com` naming convention.
- **Dual-stack endpoints**: New endpoints that are compatible with both IPv4 and IPv6 and follows the `streams-dynamodb.<region>.api.aws` naming convention.

> **Note**
>
> For a complete list of DynamoDB and DynamoDB Streams Regions and endpoints, see Regions and endpoints in the *AWS General Reference*.

The AWS SDKs provide separate clients for DynamoDB and DynamoDB Streams. Depending on your requirements, your application can access a DynamoDB endpoint, a DynamoDB Streams endpoint, or both at the same time. To connect to both endpoints, your application must instantiate two clients—one for DynamoDB and one for DynamoDB Streams.

Enabling a stream

You can enable a stream on a new table when you create it using the AWS CLI or one of the AWS SDKs. You can also enable or disable a stream on an existing table, or change the settings of a stream. DynamoDB Streams operates asynchronously, so there is no performance impact on a table if you enable a stream.

The easiest way to manage DynamoDB Streams is by using the AWS Management Console.

1. Sign in to the AWS Management Console and open the DynamoDB console at https://console.aws.amazon.com/dynamodb/.
2. On the DynamoDB console dashboard, choose **Tables** and select an existing table.
3. Choose the **Exports and streams** tab.

4. In the **DynamoDB stream details** section, choose **Turn on**.
5. On the **Turn on DynamoDB stream** page, choose the information that will be written to the stream whenever the data in the table is modified:

   - **Key attributes only** — Only the key attributes of the modified item.
   - **New image** — The entire item, as it appears after it was modified.
   - **Old image** — The entire item, as it appeared before it was modified.
   - **New and old images** — Both the new and the old images of the item.

   When the settings are as you want them, choose **Turn on stream**.
6. (Optional) To disable an existing stream, choose **Turn off** under **DynamoDB stream details**.

You can also use the `CreateTable` or `UpdateTable` API operations to enable or modify a stream. The `StreamSpecification` parameter determines how the stream is configured:

- `StreamEnabled` — Specifies whether a stream is enabled (`true`) or disabled (`false`) for the table.
- `StreamViewType` — Specifies the information that will be written to the stream whenever data in the table is modified:
  - `KEYS_ONLY` — Only the key attributes of the modified item.
  - `NEW_IMAGE` — The entire item, as it appears after it was modified.
  - `OLD_IMAGE` — The entire item, as it appeared before it was modified.
  - `NEW_AND_OLD_IMAGES` — Both the new and the old images of the item.

You can enable or disable a stream at any time. However, you receive a `ValidationException` if you try to enable a stream on a table that already has a stream. You also receive a `ValidationException` if you try to disable a stream on a table that doesn't have a stream.

When you set `StreamEnabled` to `true`, DynamoDB creates a new stream with a unique stream descriptor assigned to it. If you disable and then re-enable a stream on the table, a new stream is created with a different stream descriptor.

Every stream is uniquely identified by an Amazon Resource Name (ARN). The following is an example ARN for a stream on a DynamoDB table named `TestTable`.

```
arn:aws:dynamodb:us-west-2:111122223333:table/TestTable/stream/2015-05-11T21:21:33.291
```

To determine the latest stream descriptor for a table, issue a DynamoDB `DescribeTable` request and look for the `LatestStreamArn` element in the response.

> **Note**
>
> It is not possible to edit a `StreamViewType` once a stream has been setup. If you need to make changes to a stream after it has been setup, you must disable the current stream and create a new one.

Reading and processing a stream

To read and process a stream, your application must connect to a DynamoDB Streams endpoint and issue API requests.

A stream consists of *stream records*. Each stream record represents a single data modification in the DynamoDB table to which the stream belongs. Each stream record is assigned a sequence number, reflecting the order in which the record was published to the stream.

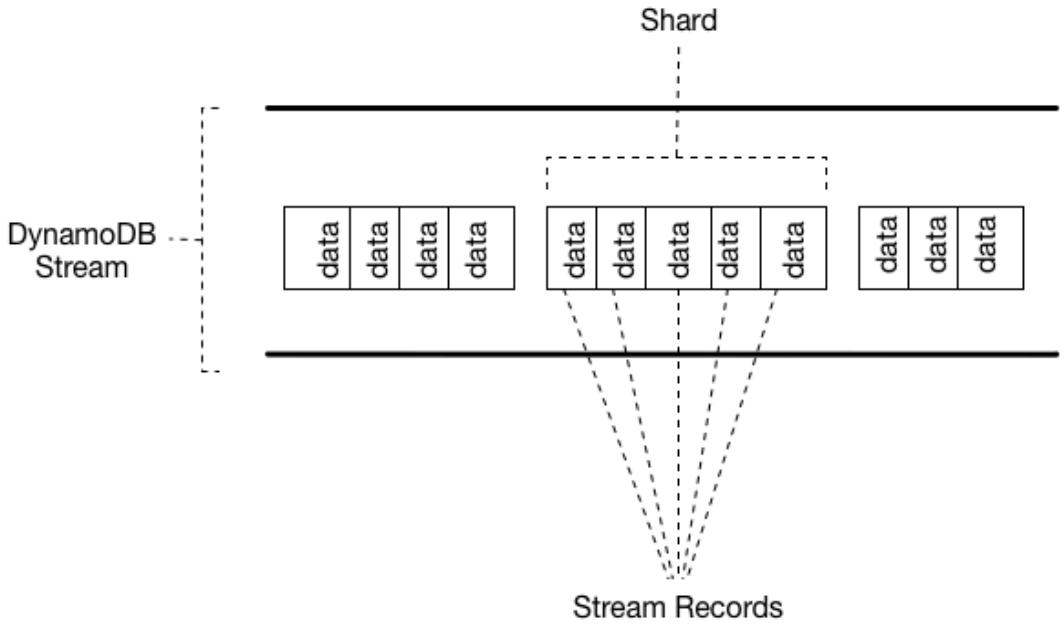
Stream records are organized into groups, or *shards*. Each shard acts as a container for multiple stream records, and contains information required for accessing and iterating through these records. The stream records within a shard are removed automatically after 24 hours.

Shards are ephemeral: They are created and deleted automatically, as needed. Any shard can also split into multiple new shards; this also occurs automatically. (It's also possible for a parent shard to have just one child shard.) A shard might split in response to high levels of write activity on its parent table, so that applications can process records from multiple shards in parallel.

If you disable a stream, any shards that are open will be closed. The data in the stream will continue to be readable for 24 hours.

Because shards have a lineage (parent and children), an application must always process a parent shard before it processes a child shard. This helps ensure that the stream records are also processed in the correct order. (If you use the DynamoDB Streams Kinesis Adapter, this is handled for you. Your application processes the shards and stream records in the correct order. It automatically handles new or expired shards, in addition to shards that split while the application is running. For more information, see Using the DynamoDB Streams Kinesis adapter to process stream records.)

The following diagram shows the relationship between a stream, shards in the stream, and stream records in the shards.



> **Note**
>
> If you perform a `PutItem` or `UpdateItem` operation that does not change any data in an item, DynamoDB Streams does *not* write a stream record for that operation.

To access a stream and process the stream records within, you must do the following:

- Determine the unique ARN of the stream that you want to access.
- Determine which shards in the stream contain the stream records that you are interested in.
- Access the shards and retrieve the stream records that you want.

> **Note**
>
> No more than two processes at most should be reading from the same stream's shard at the same time. Having more than two readers per shard can result in throttling.

The DynamoDB Streams API provides the following actions for use by application programs:

- `ListStreams` — Returns a list of stream descriptors for the current account and endpoint. You can optionally request just the stream descriptors for a particular table name.
- `DescribeStream` — Returns information about a stream, including the current status of the stream, its Amazon Resource Name (ARN), the composition of its shards, and its corresponding DynamoDB table. You can optionally use the `ShardFilter` field to retrieve the existing child shard associated with the parent shard.
- `GetShardIterator` — Returns a *shard iterator*, which describes a location within a shard. You can request that the iterator provide access to the oldest point, the newest point, or a particular point in the stream.
- `GetRecords` — Returns the stream records from within a given shard. You must provide the shard iterator returned from a `GetShardIterator` request.

For complete descriptions of these API operations, including example requests and responses, see the Amazon DynamoDB Streams API Reference.

Shard discovery

Discover new shards in your DynamoDB stream with two powerful methods. As a Amazon DynamoDB Streams user, you have two effective ways to track and identify new shards:

Polling the entire stream topology

Use the `DescribeStream` API to regularly poll the stream. This returns all shards in the stream, including any new shards that have been created. By comparing results over time, you can detect newly added shards.

Discovering child shards

Use the `DescribeStream` API with the `ShardFilter` parameter to find a subset of shards. By specifying a parent shard in the request, DynamoDB Streams will return its immediate child shards. This approach is useful when you only need to track shard lineage without scanning the entire stream.

Applications consuming data from DynamoDB Streams can efficiently transition from reading a closed shard to its child shard using this `ShardFilter` parameter, avoiding repeated calls to the `DescribeStream` API to retrieve and traverse the shard map for all closed and open shards. This helps to quickly discover child shards after a parent shard has been closed, making your stream processing applications more responsive and cost-effective.

Both methods empower you to stay on top of your DynamoDB Streams' evolving structure, ensuring you never miss critical data updates or shard modifications.

Data retention limit for DynamoDB Streams

All data in DynamoDB Streams is subject to a 24-hour lifetime. You can retrieve and analyze the last 24 hours of activity for any given table. However, data that is older than 24 hours is susceptible to trimming (removal) at any moment.

If you disable a stream on a table, the data in the stream continues to be readable for 24 hours. After this time, the data expires and the stream records are automatically deleted. There is no mechanism for manually deleting an existing stream. You must wait until the retention limit expires (24 hours), and all the stream records will be deleted.

DynamoDB Streams and Time to Live

You can back up, or otherwise process, items that are deleted by Time to Live (TTL) by enabling Amazon DynamoDB Streams on the table and processing the streams records of the expired items. For more information, see Reading and processing a stream.

The streams record contains a user identity field `Records[<index>].userIdentity`.

Items that are deleted by the Time to Live process after expiration have the following fields:

- `Records[<index>].userIdentity.type`

  `"Service"`
- `Records[<index>].userIdentity.principalId`

  `"dynamodb.amazonaws.com"`

> **Note**
>
> When you use TTL in a global table, the region the TTL was performed in will have the `userIdentity` field set. This field won't be set in other regions when the delete is replicated.

The following JSON shows the relevant portion of a single streams record.

```
"Records": [
    {
        ...

        "userIdentity": {
            "type": "Service",
            "principalId": "dynamodb.amazonaws.com"
        }

        ...

    }
]
```

Using DynamoDB Streams and Lambda to archive TTL deleted items

Combining DynamoDB Time to Live (TTL), DynamoDB Streams, and AWS Lambda can help simplify archiving data, reduce DynamoDB storage costs, and reduce code complexity. Using Lambda as the stream consumer provides many advantages, most notably the cost reduction compared to other consumers such as Kinesis Client Library (KCL). You aren't charged for `GetRecords` API calls on your DynamoDB stream when using Lambda to consume events, and Lambda can provide event filtering by identifying JSON patterns in a stream event. With event-pattern content filtering, you can define up to five different filters to control which events are sent to Lambda for processing. This helps reduce invocations of your Lambda functions, simplifies code, and reduces overall cost.

While DynamoDB Streams contains all data modifications, such as `Create`, `Modify`, and `Remove` actions, this can result in unwanted invocations of your archive Lambda function. For example, say you have a table with 2 million data modifications per hour flowing into the stream, but less than 5 percent of these are item deletes that will expire through the TTL process and need to be archived. With Lambda event source filters, the Lambda function will only invoke 100,000 times per hour. The result with event filtering is that you're charged only for the needed invocations instead of the 2 million invocations you would have without event filtering.

Event filtering is applied to the Lambda event source mapping, which is a resource that reads from a chosen event—the DynamoDB stream—and invokes a Lambda function. In the following diagram, you can see how a Time to Live deleted item is consumed by a Lambda function using streams and event filters.

DynamoDB Time to Live event filter pattern

Adding the following JSON to your event source mapping filter criteria allows invocation of your Lambda function only for TTL deleted items:

```
{
    "Filters": [
        {
            "Pattern": { "userIdentity": { "type": ["Service"], "principalId":
 ["dynamodb.amazonaws.com"] } }
        }
    ]
}
```

Create an AWS Lambda event source mapping

Use the following code snippets to create a filtered event source mapping which you can connect to a table's DynamoDB stream. Each code block includes the event filter pattern.

AWS CLI

```
aws lambda create-event-source-mapping \
--event-source-arn 'arn:aws:dynamodb:eu-west-1:012345678910:table/test/
stream/2021-12-10T00:00:00.000' \
--batch-size 10 \
--enabled \
--function-name test_func \
--starting-position LATEST \
--filter-criteria '{"Filters": [{"Pattern": "{\"userIdentity\":{\"type\":[\"Service
\"],\"principalId\":[\"dynamodb.amazonaws.com\"]}}"}]}'
```

Java

```
LambdaClient client = LambdaClient.builder()
        .region(Region.EU_WEST_1)
        .build();
```

```
Filter userIdentity = Filter.builder()
        .pattern("{\"userIdentity\":{\"type\":[\"Service\"],\"principalId\":
[\"dynamodb.amazonaws.com\"]}}")
        .build();

FilterCriteria filterCriteria = FilterCriteria.builder()
        .filters(userIdentity)
        .build();

CreateEventSourceMappingRequest mappingRequest =
 CreateEventSourceMappingRequest.builder()
        .eventSourceArn("arn:aws:dynamodb:eu-west-1:012345678910:table/test/
stream/2021-12-10T00:00:00.000")
        .batchSize(10)
        .enabled(Boolean.TRUE)
        .functionName("test_func")
        .startingPosition("LATEST")
        .filterCriteria(filterCriteria)
        .build();

try{
    CreateEventSourceMappingResponse eventSourceMappingResponse =
 client.createEventSourceMapping(mappingRequest);
    System.out.println("The mapping ARN is
 "+eventSourceMappingResponse.eventSourceArn());

}catch (ServiceException e){
    System.out.println(e.getMessage());
}
```

Node

```
const client = new LambdaClient({ region: "eu-west-1" });

const input = {
    EventSourceArn: "arn:aws:dynamodb:eu-west-1:012345678910:table/test/
stream/2021-12-10T00:00:00.000",
    BatchSize: 10,
    Enabled: true,
    FunctionName: "test_func",
    StartingPosition: "LATEST",
```

```
    FilterCriteria: { "Filters": [{ "Pattern": "{\"userIdentity\":{\"type\":
[\"Service\"],\"principalId\":[\"dynamodb.amazonaws.com\"]}}" }] }
}

const command = new CreateEventSourceMappingCommand(input);

try {
    const results = await client.send(command);
    console.log(results);
} catch (err) {
    console.error(err);
}
```

Python

```
session = boto3.session.Session(region_name = 'eu-west-1')
client = session.client('lambda')

try:
    response = client.create_event_source_mapping(
        EventSourceArn='arn:aws:dynamodb:eu-west-1:012345678910:table/test/
stream/2021-12-10T00:00:00.000',
        BatchSize=10,
        Enabled=True,
        FunctionName='test_func',
        StartingPosition='LATEST',
        FilterCriteria={
            'Filters': [
                {
                    'Pattern': "{\"userIdentity\":{\"type\":[\"Service\"],
\"principalId\":[\"dynamodb.amazonaws.com\"]}}"
                },
            ]
        }
    )
    print(response)
except Exception as e:
    print(e)
```

JSON

```
{
  "userIdentity": {
```

```
      "type": ["Service"],
      "principalId": ["dynamodb.amazonaws.com"]
    }
}
```

Using the DynamoDB Streams Kinesis adapter to process stream records

Using the Amazon Kinesis Adapter is the recommended way to consume streams from Amazon DynamoDB. The DynamoDB Streams API is intentionally similar to that of Kinesis Data Streams. In both services, data streams are composed of shards, which are containers for stream records. Both services' APIs contain `ListStreams`, `DescribeStream`, `GetShards`, and `GetShardIterator` operations. (Although these DynamoDB Streams actions are similar to their counterparts in Kinesis Data Streams, they are not 100 percent identical.)

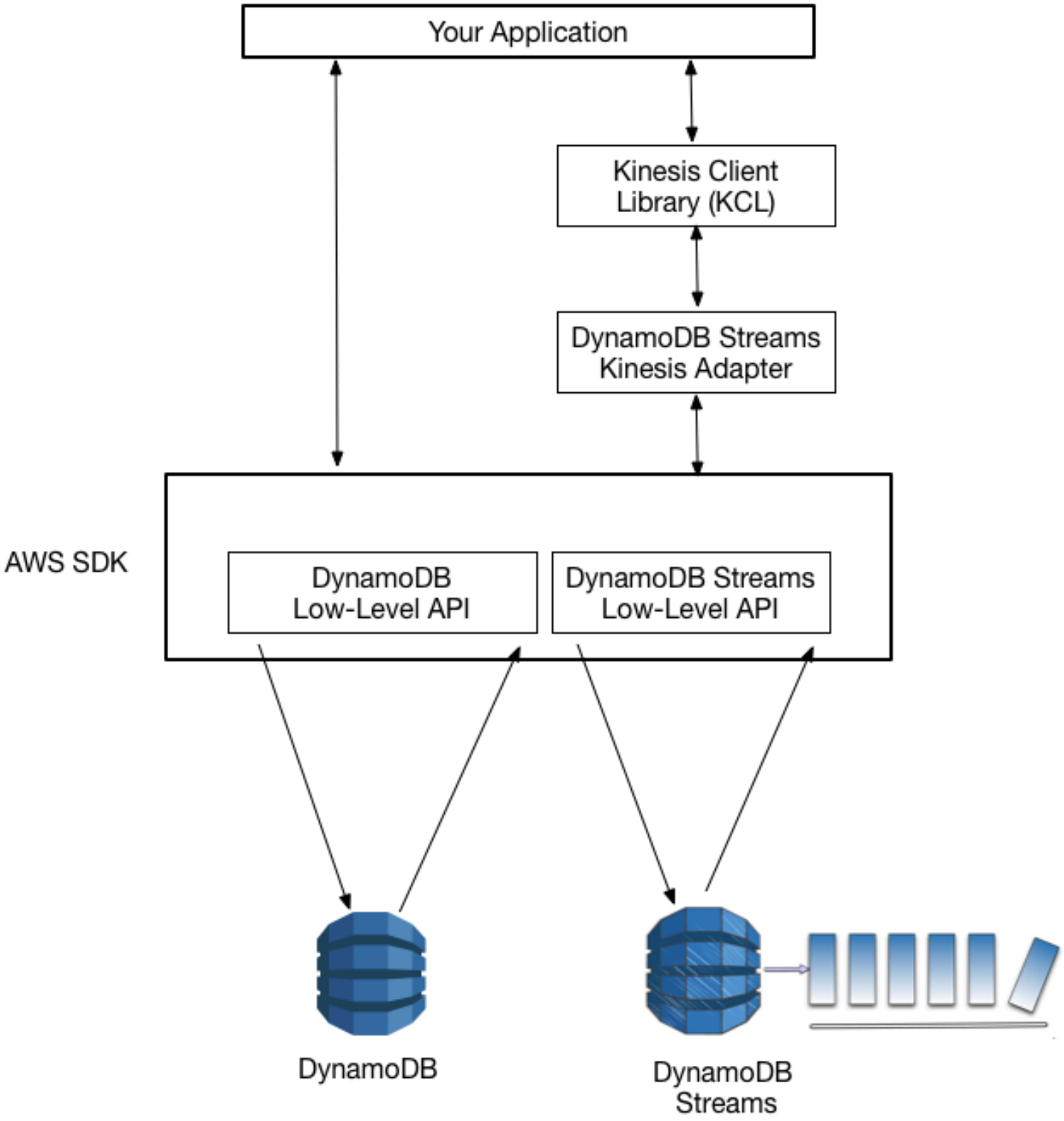
As a DynamoDB Streams user, you can use the design patterns found within the KCL to process DynamoDB Streams shards and stream records. To do this, you use the DynamoDB Streams Kinesis Adapter. The Kinesis Adapter implements the Kinesis Data Streams interface so that the KCL can be used for consuming and processing records from DynamoDB Streams. For instructions on how to set up and install the DynamoDB Streams Kinesis Adapter, see the GitHub repository.

You can write applications for Kinesis Data Streams using the Kinesis Client Library (KCL). The KCL simplifies coding by providing useful abstractions above the low-level Kinesis Data Streams API. For more information about the KCL, see the Developing consumers using the Kinesis client library in the *Amazon Kinesis Data Streams Developer Guide*.

DynamoDB recommends using KCL version 3.x with AWS SDK for Java v2.x. The current DynamoDB Streams Kinesis Adapter version 1.x with AWS SDK for AWS SDK for Java v1.x will continue to be fully supported throughout its lifecycle as intended during the transitional period in alignment with AWS SDKs and Tools maintenance policy.

> **Note**
>
> Amazon Kinesis Client Library (KCL) versions 1.x and 2.x are outdated. KCL 1.x will reach end-of-support on January 30, 2026. We strongly recommend that you migrate your KCL applications using version 1.x to the latest KCL version before January 30, 2026. To find the latest KCL version, see the Amazon Kinesis Client Library page on GitHub. For information about the latest KCL versions, see Use Kinesis Client Library. For information about migrating from KCL 1.x to KCL 3.x, see Migrating from KCL 1.x to KCL 3.x.

The following diagram shows how these libraries interact with one another.



With the DynamoDB Streams Kinesis Adapter in place, you can begin developing against the KCL interface, with the API calls seamlessly directed at the DynamoDB Streams endpoint.

When your application starts, it calls the KCL to instantiate a worker. You must provide the worker with configuration information for the application, such as the stream descriptor and AWS credentials, and the name of a record processor class that you provide. As it runs the code in the record processor, the worker performs the following tasks:

- Connects to the stream
- Enumerates the shards within the stream
- Checks and enumerates child shards of a closed parent shard within the stream
- Coordinates shard associations with other workers (if any)
- Instantiates a record processor for every shard it manages
- Pulls records from the stream
- Scales GetRecords API calling rate during high throughput (if catch-up mode is configured)
- Pushes the records to the corresponding record processor
- Checkpoints processed records
- Balances shard-worker associations when the worker instance count changes
- Balances shard-worker associations when shards are split

The KCL adapter supports catch-up mode, an automatic calling rate adjustment feature for handling temporary throughput increases. When stream processing lag exceeds a configurable threshold (default one minute), catch-up mode scales GetRecords API calling frequency by a configurable value (default 3x) to retrieve records faster, then returns to normal once the lag drops. This is valuable during high-throughput periods where DynamoDB write activity can overwhelm consumers using default polling rates. Catch-up mode can be enabled through the `catchupEnabled` configuration parameter (default false).

Note

For a description of the KCL concepts listed here, see Developing consumers using the Kinesis client library in the *Amazon Kinesis Data Streams Developer Guide*.
For more information on using streams with AWS Lambda see DynamoDB Streams and AWS Lambda triggers

Migrating from KCL 1.x to KCL 3.x

Overview

This guide provides instructions for migrating your consumer application from KCL 1.x to KCL 3.x. Due to architectural differences between KCL 1.x and KCL 3.x, migration requires updating several components to ensure compatibility.

KCL 1.x uses different classes and interfaces compared to KCL 3.x. You must migrate the record processor, record processor factory, and worker classes to the KCL 3.x compatible format first, and follow the migration steps for KCL 1.x to KCL 3.x migration.

Migration steps

Topics

- Step 1: Migrate the record processor
- Step 2: Migrate the record processor factory
- Step 3: Migrate the worker
- Step 4: KCL 3.x configuration overview and recommendations
- Step 5: Migrate from KCL 2.x to KCL 3.x

Step 1: Migrate the record processor

The following example shows a record processor implemented for KCL 1.x DynamoDB Streams Kinesis adapter:

```
package com.amazonaws.kcl;

import com.amazonaws.services.kinesis.clientlibrary.exceptions.InvalidStateException;
import com.amazonaws.services.kinesis.clientlibrary.exceptions.ShutdownException;
import
 com.amazonaws.services.kinesis.clientlibrary.interfaces.IRecordProcessorCheckpointer;
import com.amazonaws.services.kinesis.clientlibrary.interfaces.v2.IRecordProcessor;
import
 com.amazonaws.services.kinesis.clientlibrary.interfaces.v2.IShutdownNotificationAware;
import com.amazonaws.services.kinesis.clientlibrary.lib.worker.ShutdownReason;
import com.amazonaws.services.kinesis.clientlibrary.types.InitializationInput;
import com.amazonaws.services.kinesis.clientlibrary.types.ProcessRecordsInput;
import com.amazonaws.services.kinesis.clientlibrary.types.ShutdownInput;
```

```
public class StreamsRecordProcessor implements IRecordProcessor,
 IShutdownNotificationAware {
    @Override
    public void initialize(InitializationInput initializationInput) {
        //
        // Setup record processor
        //
    }

    @Override
    public void processRecords(ProcessRecordsInput processRecordsInput) {
        for (Record record : processRecordsInput.getRecords()) {
            String data = new String(record.getData().array(),
 Charset.forName("UTF-8"));
            System.out.println(data);
            if (record instanceof RecordAdapter) {
                // record processing and checkpointing logic
            }
        }
    }

    @Override
    public void shutdown(ShutdownInput shutdownInput) {
        if (shutdownInput.getShutdownReason() == ShutdownReason.TERMINATE) {
            try {
                shutdownInput.getCheckpointer().checkpoint();
            } catch (ShutdownException | InvalidStateException e) {
                throw new RuntimeException(e);
            }
        }
    }

    @Override
    public void shutdownRequested(IRecordProcessorCheckpointer checkpointer) {
        try {
            checkpointer.checkpoint();
        } catch (ShutdownException | InvalidStateException e) {
            //
            // Swallow exception
            //
            e.printStackTrace();
        }
    }
```

```
}
```

To migrate the RecordProcessor class

1. Change the interfaces from `com.amazonaws.services.kinesis.clientlibrary.interfaces.v2.IRecordProcessor` and `com.amazonaws.services.kinesis.clientlibrary.interfaces.v2.IShutdownNotificat` to `com.amazonaws.services.dynamodbv2.streamsadapter.processor.DynamoDBStreamsSha` as follows:

```
// import
 com.amazonaws.services.kinesis.clientlibrary.interfaces.v2.IRecordProcessor;
// import
 com.amazonaws.services.kinesis.clientlibrary.interfaces.v2.IShutdownNotificationAware;

import
 com.amazonaws.services.dynamodbv2.streamsadapter.processor.DynamoDBStreamsShardRecordProce
```

2. Update import statements for the `initialize` and `processRecords` methods:

```
// import com.amazonaws.services.kinesis.clientlibrary.types.InitializationInput;
import software.amazon.kinesis.lifecycle.events.InitializationInput;

// import com.amazonaws.services.kinesis.clientlibrary.types.ProcessRecordsInput;
import
 com.amazonaws.services.dynamodbv2.streamsadapter.model.DynamoDBStreamsProcessRecordsInput;
```

3. Replace the `shutdownRequested` method with the following new methods: `leaseLost`, `shardEnded`, and `shutdownRequested`.

```
//    @Override
//    public void shutdownRequested(IRecordProcessorCheckpointer checkpointer) {
//        //
//        // This is moved to shardEnded(...) and
 shutdownRequested(ShutdownReauestedInput)
//        //
//        try {
//            checkpointer.checkpoint();
//        } catch (ShutdownException | InvalidStateException e) {
```

```
//            //
//            // Swallow exception
//            //
//            e.printStackTrace();
//        }
//    }

    @Override
    public void leaseLost(LeaseLostInput leaseLostInput) {

    }

    @Override
    public void shardEnded(ShardEndedInput shardEndedInput) {
        try {
            shardEndedInput.checkpointer().checkpoint();
        } catch (ShutdownException | InvalidStateException e) {
            //
            // Swallow the exception
            //
            e.printStackTrace();
        }
    }

    @Override
    public void shutdownRequested(ShutdownRequestedInput shutdownRequestedInput) {
        try {
            shutdownRequestedInput.checkpointer().checkpoint();
        } catch (ShutdownException | InvalidStateException e) {
            //
            // Swallow the exception
            //
            e.printStackTrace();
        }
    }
```

The following is the updated version of the record processor class:

```
package com.amazonaws.codesamples;

import software.amazon.kinesis.exceptions.InvalidStateException;
import software.amazon.kinesis.exceptions.ShutdownException;
```

```
import software.amazon.kinesis.lifecycle.events.InitializationInput;
import software.amazon.kinesis.lifecycle.events.LeaseLostInput;
import
 com.amazonaws.services.dynamodbv2.streamsadapter.model.DynamoDBStreamsProcessRecordsInput;
import software.amazon.kinesis.lifecycle.events.ShardEndedInput;
import software.amazon.kinesis.lifecycle.events.ShutdownRequestedInput;
import
 software.amazon.dynamodb.streamsadapter.processor.DynamoDBStreamsShardRecordProcessor;
import
 software.amazon.dynamodb.streamsadapter.adapter.DynamoDBStreamsKinesisClientRecord;
import
 com.amazonaws.services.dynamodbv2.streamsadapter.processor.DynamoDBStreamsShardRecordProcessor
import
 com.amazonaws.services.dynamodbv2.streamsadapter.adapter.DynamoDBStreamsClientRecord;
import software.amazon.awssdk.services.dynamodb.model.Record;

public class StreamsRecordProcessor implements DynamoDBStreamsShardRecordProcessor {

    @Override
    public void initialize(InitializationInput initializationInput) {

    }

    @Override
    public void processRecords(DynamoDBStreamsProcessRecordsInput processRecordsInput)
 {
        for (DynamoDBStreamsKinesisClientRecord record: processRecordsInput.records())
            Record ddbRecord = record.getRecord();
            // processing and checkpointing logic for the ddbRecord
        }
    }

    @Override
    public void leaseLost(LeaseLostInput leaseLostInput) {

    }

    @Override
    public void shardEnded(ShardEndedInput shardEndedInput) {
        try {
            shardEndedInput.checkpointer().checkpoint();
        } catch (ShutdownException | InvalidStateException e) {
            //
            // Swallow the exception
```

```
            //
            e.printStackTrace();
        }
    }

    @Override
    public void shutdownRequested(ShutdownRequestedInput shutdownRequestedInput) {
        try {
            shutdownRequestedInput.checkpointer().checkpoint();
        } catch (ShutdownException | InvalidStateException e) {
            //
            // Swallow the exception
            //
            e.printStackTrace();
        }
    }
}
```

> **Note**
>
> DynamoDB Streams Kinesis Adapter now uses SDKv2 Record model. In SDKv2, complex `AttributeValue` objects (BS, NS, M, L, SS) never return null. Use `hasBs()`, `hasNs()`, `hasM()`, `hasL()`, `hasSs()` methods to verify if these values exist.

Step 2: Migrate the record processor factory

The record processor factory is responsible for creating record processors when a lease is acquired. The following is an example of a KCL 1.x factory:

```
package com.amazonaws.codesamples;

import software.amazon.dynamodb.AmazonDynamoDB;
import com.amazonaws.services.kinesis.clientlibrary.interfaces.v2.IRecordProcessor;
import
 com.amazonaws.services.kinesis.clientlibrary.interfaces.v2.IRecordProcessorFactory;

public class StreamsRecordProcessorFactory implements IRecordProcessorFactory {

    @Override
    public IRecordProcessor createProcessor() {
        return new StreamsRecordProcessor(dynamoDBClient, tableName);
```

```
    }
}
```

To migrate the `RecordProcessorFactory`

- Change the implemented interface from `com.amazonaws.services.kinesis.clientlibrary.interfaces.v2.IRecordProcessorFa` to `software.amazon.kinesis.processor.ShardRecordProcessorFactory`, as follows:

```
// import
 com.amazonaws.services.kinesis.clientlibrary.interfaces.v2.IRecordProcessor;
import software.amazon.kinesis.processor.ShardRecordProcessor;

// import
 com.amazonaws.services.kinesis.clientlibrary.interfaces.v2.IRecordProcessorFactory;
import software.amazon.kinesis.processor.ShardRecordProcessorFactory;

// public class TestRecordProcessorFactory implements IRecordProcessorFactory {
public class StreamsRecordProcessorFactory implements ShardRecordProcessorFactory {

Change the return signature for createProcessor.

// public IRecordProcessor createProcessor() {
public ShardRecordProcessor shardRecordProcessor() {
```

The following is an example of the record processor factory in 3.0:

```
package com.amazonaws.codesamples;

import software.amazon.kinesis.processor.ShardRecordProcessor;
import software.amazon.kinesis.processor.ShardRecordProcessorFactory;

public class StreamsRecordProcessorFactory implements ShardRecordProcessorFactory {

    @Override
    public ShardRecordProcessor shardRecordProcessor() {
        return new StreamsRecordProcessor();
    }
}
```

Step 3: Migrate the worker

In version 3.0 of the KCL, a new class, called **Scheduler**, replaces the **Worker** class. The following is an example of a KCL 1.x worker:

```
final KinesisClientLibConfiguration config = new KinesisClientLibConfiguration(...)
final IRecordProcessorFactory recordProcessorFactory = new RecordProcessorFactory();
final Worker worker = StreamsWorkerFactory.createDynamoDbStreamsWorker(
        recordProcessorFactory,
        workerConfig,
        adapterClient,
        amazonDynamoDB,
        amazonCloudWatchClient);
```

To migrate the worker

1. Change the `import` statement for the `Worker` class to the import statements for the `Scheduler` and `ConfigsBuilder` classes.

   ```
   // import com.amazonaws.services.kinesis.clientlibrary.lib.worker.Worker;
   import software.amazon.kinesis.coordinator.Scheduler;
   import software.amazon.kinesis.common.ConfigsBuilder;
   ```

2. Import `StreamTracker` and change import of `StreamsWorkerFactory` to `StreamsSchedulerFactory`.

   ```
   import software.amazon.kinesis.processor.StreamTracker;
   // import software.amazon.dynamodb.streamsadapter.StreamsWorkerFactory;
   import software.amazon.dynamodb.streamsadapter.StreamsSchedulerFactory;
   ```

3. Choose the position from which to start the application. It can be `TRIM_HORIZON` or `LATEST`.

   ```
   import software.amazon.kinesis.common.InitialPositionInStream;
   import software.amazon.kinesis.common.InitialPositionInStreamExtended;
   ```

4. Create a `StreamTracker` instance.

   ```
   StreamTracker streamTracker = StreamsSchedulerFactory.createSingleStreamTracker(
           streamArn,

    InitialPositionInStreamExtended.newInitialPosition(InitialPositionInStream.TRIM_HORIZON)
   ```

```
);
```

5. Create the `AmazonDynamoDBStreamsAdapterClient` object.

```
import software.amazon.dynamodb.streamsadapter.AmazonDynamoDBStreamsAdapterClient;
import software.amazon.awssdk.auth.credentials.AwsCredentialsProvider;
import software.amazon.awssdk.auth.credentials.DefaultCredentialsProvider;

...

AwsCredentialsProvider credentialsProvider = DefaultCredentialsProvider.create();

AmazonDynamoDBStreamsAdapterClient adapterClient = new
 AmazonDynamoDBStreamsAdapterClient(
        credentialsProvider, awsRegion);
```

6. Create the `ConfigsBuilder` object.

```
import software.amazon.kinesis.common.ConfigsBuilder;

...
ConfigsBuilder configsBuilder = new ConfigsBuilder(
                streamTracker,
                applicationName,
                adapterClient,
                dynamoDbAsyncClient,
                cloudWatchAsyncClient,
                UUID.randomUUID().toString(),
                new StreamsRecordProcessorFactory());
```

7. Create the `Scheduler` using `ConfigsBuilder` as shown in the following example:

```
import java.util.UUID;

import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbAsyncClient;
import software.amazon.awssdk.services.cloudwatch.CloudWatchAsyncClient;
import software.amazon.awssdk.services.kinesis.KinesisAsyncClient;

import software.amazon.kinesis.common.KinesisClientUtil;
import software.amazon.kinesis.coordinator.Scheduler;
```

```
...


DynamoDbAsyncClient dynamoClient =
 DynamoDbAsyncClient.builder().region(region).build();
CloudWatchAsyncClient cloudWatchClient =
 CloudWatchAsyncClient.builder().region(region).build();


DynamoDBStreamsPollingConfig pollingConfig = new
 DynamoDBStreamsPollingConfig(adapterClient);
pollingConfig.idleTimeBetweenReadsInMillis(idleTimeBetweenReadsInMillis);

// Use ConfigsBuilder to configure settings
RetrievalConfig retrievalConfig = configsBuilder.retrievalConfig();
retrievalConfig.retrievalSpecificConfig(pollingConfig);

CoordinatorConfig coordinatorConfig = configsBuilder.coordinatorConfig();
coordinatorConfig.clientVersionConfig(CoordinatorConfig.ClientVersionConfig.CLIENT_VERSION_

Scheduler scheduler = StreamsSchedulerFactory.createScheduler(
                configsBuilder.checkpointConfig(),
                coordinatorConfig,
                configsBuilder.leaseManagementConfig(),
                configsBuilder.lifecycleConfig(),
                configsBuilder.metricsConfig(),
                configsBuilder.processorConfig(),
                retrievalConfig,
                adapterClient
        );
```

Important

The `CLIENT_VERSION_CONFIG_COMPATIBLE_WITH_2X` setting maintains compatibility between DynamoDB Streams Kinesis Adapter for KCL v3 and KCL v1, not between KCL v2 and v3.

Step 4: KCL 3.x configuration overview and recommendations

For a detailed description of the configurations introduced post KCL 1.x that are relevant in KCL 3.x see KCL configurations and KCL migration client configuration.

> **Important**
>
> Instead of directly creating objects of `checkpointConfig`, `coordinatorConfig`, `leaseManagementConfig`, `metricsConfig`, `processorConfig` and `retrievalConfig`, we recommend using `ConfigsBuilder` to set configurations in KCL 3.x and later versions to avoid Scheduler initialization issues. `ConfigsBuilder` provides a more flexible and maintainable way to configure your KCL application.

Configurations with update default value in KCL 3.x

`billingMode`

In KCL version 1.x, the default value for `billingMode` is set to `PROVISIONED`. However, with KCL version 3.x, the default `billingMode` is `PAY_PER_REQUEST` (on-demand mode). We recommend that you use the on-demand capacity mode for your lease table to automatically adjust the capacity based on your usage. For guidance on using provisioned capacity for your lease tables, see Best practices for the lease table with provisioned capacity mode.

`idleTimeBetweenReadsInMillis`

In KCL version 1.x, the default value for `idleTimeBetweenReadsInMillis` is set to is 1,000 (or 1 second). KCL version 3.x sets the default value for `idleTimeBetweenReadsInMillis` to 1,500 (or 1.5 seconds), but Amazon DynamoDB Streams Kinesis Adapter overrides the default value to 1,000 (or 1 second).

New configurations in KCL 3.x

`leaseAssignmentIntervalMillis`

This configuration defines the time interval before newly discovered shards begin processing, and is calculated as 1.5 × `leaseAssignmentIntervalMillis`. If this setting isn't explicitly configured, the time interval defaults to 1.5 × `failoverTimeMillis`. Processing new shards involves scanning the lease table and querying a global secondary index (GSI) on the lease

table. Lowering the `leaseAssignmentIntervalMillis` increases the frequency of these scan and query operations, resulting in higher DynamoDB costs. We recommend setting this value to 2000 (or 2 seconds) to minimize the delay in processing new shards.

`shardConsumerDispatchPollIntervalMillis`

This configuration defines the interval between successive polls by the shard consumer to trigger state transitions. In KCL version 1.x, this behavior was controlled by the `idleTimeInMillis` parameter, which was not exposed as a configurable setting. With KCL version 3.x, we recommend setting this config to match the value used for `idleTimeInMillis` in your KCL version 1.x setup.

Step 5: Migrate from KCL 2.x to KCL 3.x

To ensure a smooth transition and compatibility with the latest Kinesis Client Library (KCL) version, follow steps 5-8 in the migration guide's instructions for upgrading from KCL 2.x to KCL 3.x.

For common KCL 3.x troubleshooting issues, see Troubleshooting KCL consumer applications.

Roll back to the previous KCL version

This topic explains how to roll back your consumer application to the previous KCL version. The roll-back process consists of two steps:

1. Run the KCL Migration Tool.
2. Redeploy previous KCL version code.

Step 1: Run the KCL Migration Tool

When you need to roll back to the previous KCL version, you must run the KCL Migration Tool. The tool performs two important tasks:

- It removes a metadata table called worker metrics table and global secondary index on the lease table in DynamoDB. These artifacts are created by KCL 3.x but aren't needed when you roll back to the previous version.
- It makes all workers run in a mode compatible with KCL 1.x and start using the load balancing algorithm used in previous KCL versions. If you have issues with the new load balancing algorithm in KCL 3.x, this will mitigate the issue immediately.

> **Important**
>
> The coordinator state table in DynamoDB must exist and must not be deleted during the migration, rollback, and rollforward process.

> **Note**
>
> It's important that all workers in your consumer application use the same load balancing algorithm at a given time. The KCL Migration Tool makes sure that all workers in your KCL 3.x consumer application switch to the KCL 1.x compatible mode so that all workers run the same load balancing algorithm during the application rollback to the previous KCL version.

You can download the KCL Migration Tool in the scripts directory of the KCL GitHub repository. Run the script from a worker or host with appropriate permissions to write to the coordinator state table, worker metrics table, and lease table. Ensure the appropriate IAM permissions are configured for KCL consumer applications. Run the script only once per KCL application using the specified command:

```
python3 ./KclMigrationTool.py --region region --mode rollback [--
application_name applicationName] [--lease_table_name leaseTableName]
 [--coordinator_state_table_name coordinatorStateTableName] [--
worker_metrics_table_name workerMetricsTableName]
```

Parameters

`--region`

Replace *region* with your AWS Region.

`--application_name`

This parameter is required if you're using default names for your DynamoDB metadata tables (lease table, coordinator state table, and worker metrics table). If you have specified custom names for these tables, you can omit this parameter. Replace *applicationName* with your actual KCL application name. The tool uses this name to derive the default table names if custom names are not provided.

`--lease_table_name`

This parameter is needed when you have set a custom name for the lease table in your KCL configuration. If you're using the default table name, you can omit this parameter. Replace *`leaseTableName`* with the custom table name you specified for your lease table.

`--coordinator_state_table_name`

This parameter is needed when you have set a custom name for the coordinator state table in your KCL configuration. If you're using the default table name, you can omit this parameter. Replace *`coordinatorStateTableName`* with the custom table name you specified for your coordinator state table.

`--worker_metrics_table_name`

This parameter is needed when you have set a custom name for the worker metrics table in your KCL configuration. If you're using the default table name, you can omit this parameter. Replace *`workerMetricsTableName`* with the custom table name you specified for your worker metrics table.

Step 2: Redeploy the code with the previous KCL version

> **Important**
>
> Any mention of version 2.x in the output generated by the KCL Migration Tool should be interpreted as referring to KCL version 1.x. Running the script does not perform a complete rollback, it only switches the load balancing algorithm to the one used in KCL version 1.x.

After running the KCL Migration Tool for a rollback, you'll see one of these messages:

Message 1

"Rollback completed. Your application was running 2x compatible functionality. Please rollback to your previous application binaries by deploying the code with your previous KCL version."

Required action: This means that your workers were running in the KCL 1.x compatible mode. Redeploy the code with the previous KCL version to your workers.

Message 2

"Rollback completed. Your KCL Application was running 3x functionality and will rollback to 2x compatible functionality. If you don't see mitigation after a short period of time, please rollback to your previous application binaries by deploying the code with your previous KCL version."

Required action: This means that your workers were running in KCL 3.x mode and the KCL Migration Tool switched all workers to KCL 1.x compatible mode. Redeploy the code with the previous KCL version to your workers.

Message 3

"Application was already rolled back. Any KCLv3 resources that could be deleted were cleaned up to avoid charges until the application can be rolled forward with migration."

Required action: This means that your workers were already rolled back to run in the KCL 1.x compatible mode. Redeploy the code with the previous KCL version to your workers.

Roll forward to KCL 3.x after a rollback

This topic explains how to roll forward your consumer application to KCL 3.x after a rollback. When you need to roll forward, you must complete a two-step process:

1. Run the KCL Migration Tool.
2. Deploy the code with KCL 3.x.

Step 1: Run the KCL Migration Tool

Run the KCL Migration Tool with the following command to roll forward to KCL 3.x:

```
python3 ./KclMigrationTool.py --region region --mode rollforward [--
application_name applicationName] [--
coordinator_state_table_name coordinatorStateTableName]
```

Parameters

`--region`

Replace *region* with your AWS Region.

`--application_name`

This parameter is required if you're using default names for your coordinator state table. If you have specified custom names for the coordinator state table, you can omit this parameter. Replace *`applicationName`* with your actual KCL application name. The tool uses this name to derive the default table names if custom names are not provided.

`--coordinator_state_table_name`

This parameter is needed when you have set a custom name for the coordinator state table in your KCL configuration. If you're using the default table name, you can omit this parameter. Replace *`coordinatorStateTableName`* with the custom table name you specified for your coordinator state table.

After you run the migration tool in roll-forward mode, KCL creates the following DynamoDB resources required for KCL 3.x:

- A Global Secondary Index on the lease table
- A worker metrics table

Step 2: Deploy the code with KCL 3.x

After running the KCL Migration Tool for a roll forward, deploy your code with KCL 3.x to your workers. To complete your migration, see Step 8: Complete the migration.

Walkthrough: DynamoDB Streams Kinesis adapter

This section is a walkthrough of a Java application that uses the Amazon Kinesis Client Library and the Amazon DynamoDB Streams Kinesis Adapter. The application shows an example of data replication, in which write activity from one table is applied to a second table, with both tables' contents staying in sync. For the source code, see Complete program: DynamoDB Streams Kinesis adapter.

The program does the following:

1. Creates two DynamoDB tables named `KCL-Demo-src` and `KCL-Demo-dst`. Each of these tables has a stream enabled on it.
2. Generates update activity in the source table by adding, updating, and deleting items. This causes data to be written to the table's stream.

3. Reads the records from the stream, reconstructs them as DynamoDB requests, and applies the requests to the destination table.
4. Scans the source and destination tables to ensure that their contents are identical.
5. Cleans up by deleting the tables.

These steps are described in the following sections, and the complete application is shown at the end of the walkthrough.

Topics

- Step 1: Create DynamoDB tables
- Step 2: Generate update activity in source table
- Step 3: Process the stream
- Step 4: Ensure that both tables have identical contents
- Step 5: Clean up
- Complete program: DynamoDB Streams Kinesis adapter

Step 1: Create DynamoDB tables

The first step is to create two DynamoDB tables—a source table and a destination table. The `StreamViewType` on the source table's stream is `NEW_IMAGE`. This means that whenever an item is modified in this table, the item's "after" image is written to the stream. In this way, the stream keeps track of all write activity on the table.

The following example shows the code that is used for creating both tables.

```
java.util.List<AttributeDefinition> attributeDefinitions = new
 ArrayList<AttributeDefinition>();
attributeDefinitions.add(new
 AttributeDefinition().withAttributeName("Id").withAttributeType("N"));

java.util.List<KeySchemaElement> keySchema = new ArrayList<KeySchemaElement>();
keySchema.add(new
 KeySchemaElement().withAttributeName("Id").withKeyType(KeyType.HASH)); // Partition

  // key
```

```
ProvisionedThroughput provisionedThroughput = new
 ProvisionedThroughput().withReadCapacityUnits(2L)
    .withWriteCapacityUnits(2L);

StreamSpecification streamSpecification = new StreamSpecification();
streamSpecification.setStreamEnabled(true);
streamSpecification.setStreamViewType(StreamViewType.NEW_IMAGE);
CreateTableRequest createTableRequest = new
 CreateTableRequest().withTableName(tableName)
    .withAttributeDefinitions(attributeDefinitions).withKeySchema(keySchema)

 .withProvisionedThroughput(provisionedThroughput).withStreamSpecification(streamSpecification)
```

Step 2: Generate update activity in source table

The next step is to generate some write activity on the source table. While this activity is taking place, the source table's stream is also updated in near-real time.

The application defines a helper class with methods that call the `PutItem`, `UpdateItem`, and `DeleteItem` API operations for writing the data. The following code example shows how these methods are used.

```
StreamsAdapterDemoHelper.putItem(dynamoDBClient, tableName, "101", "test1");
StreamsAdapterDemoHelper.updateItem(dynamoDBClient, tableName, "101", "test2");
StreamsAdapterDemoHelper.deleteItem(dynamoDBClient, tableName, "101");
StreamsAdapterDemoHelper.putItem(dynamoDBClient, tableName, "102", "demo3");
StreamsAdapterDemoHelper.updateItem(dynamoDBClient, tableName, "102", "demo4");
StreamsAdapterDemoHelper.deleteItem(dynamoDBClient, tableName, "102");
```

Step 3: Process the stream

Now the program begins processing the stream. The DynamoDB Streams Kinesis Adapter acts as a transparent layer between the KCL and the DynamoDB Streams endpoint, so that the code can fully use KCL rather than having to make low-level DynamoDB Streams calls. The program performs the following tasks:

- It defines a record processor class, `StreamsRecordProcessor`, with methods that comply with the KCL interface definition: `initialize`, `processRecords`, and `shutdown`. The `processRecords` method contains the logic required for reading from the source table's stream and writing to the destination table.

- It defines a class factory for the record processor class (`StreamsRecordProcessorFactory`). This is required for Java programs that use the KCL.
- It instantiates a new KCL `Worker`, which is associated with the class factory.
- It shuts down the `Worker` when record processing is complete.

Optionally, enable catch-up mode in your Streams KCL Adapter configuration to automatically scale GetRecords API calling rate by 3x (default) when stream processing lag exceeds one minute (default), helping your stream consumer handle high throughput spikes in your table.

To learn more about the KCL interface definition, see Developing consumers using the Kinesis client library in the *Amazon Kinesis Data Streams Developer Guide*.

The following code example shows the main loop in `StreamsRecordProcessor`. The `case` statement determines what action to perform, based on the `OperationType` that appears in the stream record.

```
for (Record record : records) {
    String data = new String(record.getData().array(), Charset.forName("UTF-8"));
    System.out.println(data);
    if (record instanceof RecordAdapter) {
                software.amazon.dynamodb.model.Record streamRecord = ((RecordAdapter)
 record)
                    .getInternalObject();

                switch (streamRecord.getEventName()) {
                    case "INSERT":
                    case "MODIFY":
                        StreamsAdapterDemoHelper.putItem(dynamoDBClient, tableName,
                            streamRecord.getDynamodb().getNewImage());
                        break;
                    case "REMOVE":
                        StreamsAdapterDemoHelper.deleteItem(dynamoDBClient, tableName,
                            streamRecord.getDynamodb().getKeys().get("Id").getN());
                }
    }
    checkpointCounter += 1;
    if (checkpointCounter % 10 == 0) {
        try {
            checkpointer.checkpoint();
        }
```

```
        catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Step 4: Ensure that both tables have identical contents

At this point, the source and destination tables' contents are in sync. The application issues `Scan` requests against both tables to verify that their contents are, in fact, identical.

The `DemoHelper` class contains a `ScanTable` method that calls the low-level `Scan` API. The following example shows how this is used.

```
if (StreamsAdapterDemoHelper.scanTable(dynamoDBClient, srcTable).getItems()
    .equals(StreamsAdapterDemoHelper.scanTable(dynamoDBClient, destTable).getItems()))
 {
    System.out.println("Scan result is equal.");
}
else {
    System.out.println("Tables are different!");
}
```

Step 5: Clean up

The demo is complete, so the application deletes the source and destination tables. See the following code example. Even after the tables are deleted, their streams remain available for up to 24 hours, after which they are automatically deleted.

```
dynamoDBClient.deleteTable(new DeleteTableRequest().withTableName(srcTable));
dynamoDBClient.deleteTable(new DeleteTableRequest().withTableName(destTable));
```

Complete program: DynamoDB Streams Kinesis adapter

The following is the complete Java program that performs the tasks described in Walkthrough: DynamoDB Streams Kinesis adapter. When you run it, you should see output similar to the following.

```
Creating table KCL-Demo-src
```

```
Creating table KCL-Demo-dest
Table is active.
Creating worker for stream: arn:aws:dynamodb:us-west-2:111122223333:table/KCL-Demo-src/
stream/2015-05-19T22:48:56.601
Starting worker...
Scan result is equal.
Done.
```

Important

To run this program, ensure that the client application has access to DynamoDB and Amazon CloudWatch using policies. For more information, see Identity-based policies for DynamoDB.

The source code consists of four `.java` files. To build this program, add the following dependency, which includes the Amazon Kinesis Client Library (KCL) 3.x and AWS SDK for Java v2 as transitive dependencies:

Maven

```
<dependency>
    <groupId>com.amazonaws</groupId>
    <artifactId>dynamodb-streams-kinesis-adapter</artifactId>
    <version>2.1.0</version>
</dependency>
```

Gradle

```
implementation 'com.amazonaws:dynamodb-streams-kinesis-adapter:2.1.0'
```

The source files are:

- `StreamsAdapterDemo.java`
- `StreamsRecordProcessor.java`
- `StreamsRecordProcessorFactory.java`
- `StreamsAdapterDemoHelper.java`

StreamsAdapterDemo.java

```
package com.amazonaws.codesamples;

import
 com.amazonaws.services.dynamodbv2.streamsadapter.AmazonDynamoDBStreamsAdapterClient;
import com.amazonaws.services.dynamodbv2.streamsadapter.StreamsSchedulerFactory;
import
 com.amazonaws.services.dynamodbv2.streamsadapter.polling.DynamoDBStreamsPollingConfig;
import software.amazon.awssdk.auth.credentials.AwsCredentialsProvider;
import software.amazon.awssdk.auth.credentials.DefaultCredentialsProvider;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.cloudwatch.CloudWatchAsyncClient;
import software.amazon.awssdk.services.dynamodb.DynamoDbAsyncClient;
import software.amazon.awssdk.services.dynamodb.model.DeleteTableRequest;
import software.amazon.awssdk.services.dynamodb.model.DescribeTableResponse;
import software.amazon.kinesis.common.ConfigsBuilder;
import software.amazon.kinesis.common.InitialPositionInStream;
import software.amazon.kinesis.common.InitialPositionInStreamExtended;
import software.amazon.kinesis.coordinator.Scheduler;
import software.amazon.kinesis.processor.ShardRecordProcessorFactory;
import software.amazon.kinesis.processor.StreamTracker;
import software.amazon.kinesis.retrieval.RetrievalConfig;

public class StreamsAdapterDemo {

    private static DynamoDbAsyncClient dynamoDbAsyncClient;
    private static CloudWatchAsyncClient cloudWatchAsyncClient;
    private static AmazonDynamoDBStreamsAdapterClient
 amazonDynamoDbStreamsAdapterClient;

    private static String tablePrefix = "KCL-Demo";
    private static String streamArn;

    private static Region region = Region.US_EAST_1;
    private static AwsCredentialsProvider credentialsProvider =
 DefaultCredentialsProvider.create();

    public static void main( String[] args ) throws Exception {
        System.out.println("Starting demo...");
        dynamoDbAsyncClient = DynamoDbAsyncClient.builder()
                .credentialsProvider(credentialsProvider)
                .region(region)
                .build();
```

```
        cloudWatchAsyncClient = CloudWatchAsyncClient.builder()
                .credentialsProvider(credentialsProvider)
                .region(region)
                .build();
        amazonDynamoDbStreamsAdapterClient = new
AmazonDynamoDBStreamsAdapterClient(credentialsProvider, region);

        String srcTable = tablePrefix + "-src";
        String destTable = tablePrefix + "-dest";

        setUpTables();

        StreamTracker streamTracker =
StreamsSchedulerFactory.createSingleStreamTracker(streamArn,

InitialPositionInStreamExtended.newInitialPosition(InitialPositionInStream.TRIM_HORIZON));

        ShardRecordProcessorFactory shardRecordProcessorFactory =
                new StreamsAdapterDemoProcessorFactory(dynamoDbAsyncClient, destTable);

        ConfigsBuilder configsBuilder = new ConfigsBuilder(
                streamTracker,
                "streams-adapter-demo",
                amazonDynamoDbStreamsAdapterClient,
                dynamoDbAsyncClient,
                cloudWatchAsyncClient,
                "streams-demo-worker",
                shardRecordProcessorFactory
        );

        DynamoDBStreamsPollingConfig pollingConfig = new
DynamoDBStreamsPollingConfig(amazonDynamoDbStreamsAdapterClient);
        RetrievalConfig retrievalConfig = configsBuilder.retrievalConfig();
        retrievalConfig.retrievalSpecificConfig(pollingConfig);

        System.out.println("Creating scheduler for stream " + streamArn);
        Scheduler scheduler = StreamsSchedulerFactory.createScheduler(
                configsBuilder.checkpointConfig(),
                configsBuilder.coordinatorConfig(),
                configsBuilder.leaseManagementConfig(),
                configsBuilder.lifecycleConfig(),
                configsBuilder.metricsConfig(),
                configsBuilder.processorConfig(),
                retrievalConfig,
```

```
                amazonDynamoDbStreamsAdapterClient
        );

        System.out.println("Starting scheduler...");
        Thread t = new Thread(scheduler);
        t.start();

        Thread.sleep(250000);

        System.out.println("Stopping scheduler...");
        scheduler.shutdown();
        t.join();

        if (StreamsAdapterDemoHelper.scanTable(dynamoDbAsyncClient, srcTable).items()
                .equals(StreamsAdapterDemoHelper.scanTable(dynamoDbAsyncClient,
destTable).items())) {
            System.out.println("Scan result is equal.");
        } else {
            System.out.println("Tables are different!");
        }

        System.out.println("Done.");
        cleanupAndExit(0);
    }

    private static void setUpTables() {
        String srcTable = tablePrefix + "-src";
        String destTable = tablePrefix + "-dest";
        streamArn = StreamsAdapterDemoHelper.createTable(dynamoDbAsyncClient,
srcTable);
        StreamsAdapterDemoHelper.createTable(dynamoDbAsyncClient, destTable);

        awaitTableCreation(srcTable);

        performOps(srcTable);
    }

    private static void awaitTableCreation(String tableName) {
        Integer retries = 0;
        Boolean created = false;
        while (!created && retries < 100) {
            DescribeTableResponse result =
StreamsAdapterDemoHelper.describeTable(dynamoDbAsyncClient, tableName);
            created = result.table().tableStatusAsString().equals("ACTIVE");
```

```
            if (created) {
                System.out.println("Table is active.");
                return;
            } else {
                retries++;
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    // do nothing
                }
            }
        }
        System.out.println("Timeout after table creation. Exiting...");
        cleanupAndExit(1);
    }

    private static void performOps(String tableName) {
        StreamsAdapterDemoHelper.putItem(dynamoDbAsyncClient, tableName, "101",
 "test1");
        StreamsAdapterDemoHelper.updateItem(dynamoDbAsyncClient, tableName, "101",
 "test2");
        StreamsAdapterDemoHelper.deleteItem(dynamoDbAsyncClient, tableName, "101");
        StreamsAdapterDemoHelper.putItem(dynamoDbAsyncClient, tableName, "102",
 "demo3");
        StreamsAdapterDemoHelper.updateItem(dynamoDbAsyncClient, tableName, "102",
 "demo4");
        StreamsAdapterDemoHelper.deleteItem(dynamoDbAsyncClient, tableName, "102");
    }

    private static void cleanupAndExit(Integer returnValue) {
        String srcTable = tablePrefix + "-src";
        String destTable = tablePrefix + "-dest";

 dynamoDbAsyncClient.deleteTable(DeleteTableRequest.builder().tableName(srcTable).build());

 dynamoDbAsyncClient.deleteTable(DeleteTableRequest.builder().tableName(destTable).build());
        System.exit(returnValue);
    }
}
```

StreamsRecordProcessor.java

```
package com.amazonaws.codesamples;
```

```
import
 com.amazonaws.services.dynamodbv2.streamsadapter.adapter.DynamoDBStreamsClientRecord;
import
 com.amazonaws.services.dynamodbv2.streamsadapter.model.DynamoDBStreamsProcessRecordsInput;
import
 com.amazonaws.services.dynamodbv2.streamsadapter.processor.DynamoDBStreamsShardRecordProcessor
import software.amazon.awssdk.services.dynamodb.DynamoDbAsyncClient;
import software.amazon.awssdk.services.dynamodb.model.Record;
import software.amazon.kinesis.exceptions.InvalidStateException;
import software.amazon.kinesis.exceptions.ShutdownException;
import software.amazon.kinesis.lifecycle.events.InitializationInput;
import software.amazon.kinesis.lifecycle.events.LeaseLostInput;
import software.amazon.kinesis.lifecycle.events.ShardEndedInput;
import software.amazon.kinesis.lifecycle.events.ShutdownRequestedInput;

import java.nio.charset.Charset;
import java.nio.charset.StandardCharsets;

public class StreamsRecordProcessor implements DynamoDBStreamsShardRecordProcessor {

    private Integer checkpointCounter;

    private final DynamoDbAsyncClient dynamoDbAsyncClient;
    private final String tableName;

    public StreamsRecordProcessor(DynamoDbAsyncClient dynamoDbAsyncClient, String
 tableName) {
        this.dynamoDbAsyncClient = dynamoDbAsyncClient;
        this.tableName = tableName;
    }

    @Override
    public void initialize(InitializationInput initializationInput) {
        this.checkpointCounter = 0;
    }

    @Override
    public void processRecords(DynamoDBStreamsProcessRecordsInput
 dynamoDBStreamsProcessRecordsInput) {
        for (DynamoDBStreamsClientRecord record:
 dynamoDBStreamsProcessRecordsInput.records()) {
            String data = new String(record.data().array(), StandardCharsets.UTF_8);
            System.out.println(data);
```

```
            Record streamRecord = record.getRecord();

            switch (streamRecord.eventName()) {
                case INSERT:
                case MODIFY:
                    StreamsAdapterDemoHelper.putItem(dynamoDbAsyncClient, tableName,
                            streamRecord.dynamodb().newImage());
                case REMOVE:
                    StreamsAdapterDemoHelper.deleteItem(dynamoDbAsyncClient, tableName,
                            streamRecord.dynamodb().keys().get("Id").n());
            }
            checkpointCounter += 1;
            if (checkpointCounter % 10 == 0) {
                try {
                    dynamoDBStreamsProcessRecordsInput.checkpointer().checkpoint();
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        }
    }

    @Override
    public void leaseLost(LeaseLostInput leaseLostInput) {
        System.out.println("Lease Lost");
    }

    @Override
    public void shardEnded(ShardEndedInput shardEndedInput) {
        try {
            shardEndedInput.checkpointer().checkpoint();
        } catch (ShutdownException | InvalidStateException e) {
            e.printStackTrace();
        }
    }

    @Override
    public void shutdownRequested(ShutdownRequestedInput shutdownRequestedInput) {
        try {
            shutdownRequestedInput.checkpointer().checkpoint();
        } catch (ShutdownException | InvalidStateException e) {
            e.printStackTrace();
        }
    }
```

```
}
```

StreamsRecordProcessorFactory.java

```
package com.amazonaws.codesamples;

import software.amazon.awssdk.services.dynamodb.DynamoDbAsyncClient;
import software.amazon.kinesis.processor.ShardRecordProcessor;
import software.amazon.kinesis.processor.ShardRecordProcessorFactory;

public class StreamsAdapterDemoProcessorFactory implements ShardRecordProcessorFactory
 {
    private final String tableName;
    private final DynamoDbAsyncClient dynamoDbAsyncClient;

    public StreamsAdapterDemoProcessorFactory(DynamoDbAsyncClient asyncClient, String
 tableName) {
        this.tableName = tableName;
        this.dynamoDbAsyncClient = asyncClient;
    }

    @Override
    public ShardRecordProcessor shardRecordProcessor() {
        return new StreamsRecordProcessor(dynamoDbAsyncClient, tableName);
    }
}
```

StreamsAdapterDemoHelper.java

```
package com.amazonaws.codesamples;

import software.amazon.awssdk.services.dynamodb.DynamoDbAsyncClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeDefinition;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.BillingMode;
import software.amazon.awssdk.services.dynamodb.model.CreateTableRequest;
import software.amazon.awssdk.services.dynamodb.model.CreateTableResponse;
import software.amazon.awssdk.services.dynamodb.model.DeleteItemRequest;
import software.amazon.awssdk.services.dynamodb.model.DescribeTableRequest;
import software.amazon.awssdk.services.dynamodb.model.DescribeTableResponse;
import software.amazon.awssdk.services.dynamodb.model.KeySchemaElement;
import software.amazon.awssdk.services.dynamodb.model.KeyType;
import software.amazon.awssdk.services.dynamodb.model.OnDemandThroughput;
```

```
import software.amazon.awssdk.services.dynamodb.model.PutItemRequest;
import software.amazon.awssdk.services.dynamodb.model.ResourceInUseException;
import software.amazon.awssdk.services.dynamodb.model.ScanRequest;
import software.amazon.awssdk.services.dynamodb.model.ScanResponse;
import software.amazon.awssdk.services.dynamodb.model.StreamSpecification;
import software.amazon.awssdk.services.dynamodb.model.StreamViewType;
import software.amazon.awssdk.services.dynamodb.model.UpdateItemRequest;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

public class StreamsAdapterDemoHelper {

    /**
     * @return StreamArn
     */
    public static String createTable(DynamoDbAsyncClient client, String tableName) {
        List<AttributeDefinition> attributeDefinitions = new ArrayList<>();
        attributeDefinitions.add(AttributeDefinition.builder()
                .attributeName("Id")
                .attributeType("N")
                .build());

        List<KeySchemaElement> keySchema = new ArrayList<>();
        keySchema.add(KeySchemaElement.builder()
                .attributeName("Id")
                .keyType(KeyType.HASH) // Partition key
                .build());

        StreamSpecification streamSpecification = StreamSpecification.builder()
                .streamEnabled(true)
                .streamViewType(StreamViewType.NEW_IMAGE)
                .build();

        CreateTableRequest createTableRequest = CreateTableRequest.builder()
                .tableName(tableName)
                .attributeDefinitions(attributeDefinitions)
                .keySchema(keySchema)
                .billingMode(BillingMode.PAY_PER_REQUEST)
                .streamSpecification(streamSpecification)
                .build();
```

```
        try {
            System.out.println("Creating table " + tableName);
            CreateTableResponse result = client.createTable(createTableRequest).join();
            return result.tableDescription().latestStreamArn();
        } catch (Exception e) {
            if (e.getCause() instanceof ResourceInUseException) {
                System.out.println("Table already exists.");
                return describeTable(client, tableName).table().latestStreamArn();
            }
            throw e;
        }
    }

    public static DescribeTableResponse describeTable(DynamoDbAsyncClient client,
String tableName) {
        return client.describeTable(DescribeTableRequest.builder()
                        .tableName(tableName)
                        .build())
                .join();
    }

    public static ScanResponse scanTable(DynamoDbAsyncClient dynamoDbClient, String
tableName) {
        return dynamoDbClient.scan(ScanRequest.builder()
                        .tableName(tableName)
                        .build())
                .join();
    }

    public static void putItem(DynamoDbAsyncClient dynamoDbClient, String tableName,
String id, String val) {
        Map<String, AttributeValue> item = new HashMap<>();
        item.put("Id", AttributeValue.builder().n(id).build());
        item.put("attribute-1", AttributeValue.builder().s(val).build());

        putItem(dynamoDbClient, tableName, item);
    }

    public static void putItem(DynamoDbAsyncClient dynamoDbClient, String tableName,
                               Map<String, AttributeValue> items) {
        PutItemRequest putItemRequest = PutItemRequest.builder()
                .tableName(tableName)
                .item(items)
                .build();
```

```
        dynamoDbClient.putItem(putItemRequest).join();
    }

    public static void updateItem(DynamoDbAsyncClient dynamoDbClient, String tableName,
 String id, String val) {
        Map<String, AttributeValue> key = new HashMap<>();
        key.put("Id", AttributeValue.builder().n(id).build());

        Map<String, String> expressionAttributeNames = new HashMap<>();
        expressionAttributeNames.put("#attr2", "attribute-2");

        Map<String, AttributeValue> expressionAttributeValues = new HashMap<>();
        expressionAttributeValues.put(":val", AttributeValue.builder().s(val).build());

        UpdateItemRequest updateItemRequest = UpdateItemRequest.builder()
                .tableName(tableName)
                .key(key)
                .updateExpression("SET #attr2 = :val")
                .expressionAttributeNames(expressionAttributeNames)
                .expressionAttributeValues(expressionAttributeValues)
                .build();

        dynamoDbClient.updateItem(updateItemRequest).join();
    }

    public static void deleteItem(DynamoDbAsyncClient dynamoDbClient, String tableName,
 String id) {
        Map<String, AttributeValue> key = new HashMap<>();
        key.put("Id", AttributeValue.builder().n(id).build());

        DeleteItemRequest deleteItemRequest = DeleteItemRequest.builder()
                .tableName(tableName)
                .key(key)
                .build();
        dynamoDbClient.deleteItem(deleteItemRequest).join();
    }
}
```

DynamoDB Streams low-level API: Java example

> **Note**
>
> The code on this page is not exhaustive and does not handle all scenarios for consuming Amazon DynamoDB Streams. The recommended way to consume stream records from DynamoDB is through the Amazon Kinesis Adapter using the Kinesis Client Library (KCL), as described in Using the DynamoDB Streams Kinesis adapter to process stream records.

This section contains a Java program that shows DynamoDB Streams in action. The program does the following:

1. Creates a DynamoDB table with a stream enabled.
2. Describes the stream settings for this table.
3. Modifies data in the table.
4. Describes the shards in the stream.
5. Reads the stream records from the shards.
6. Fetches child shards and continues reading records.
7. Cleans up.

When you run the program, you will see output similar to the following.

```
Testing Streams Demo
Creating an Amazon DynamoDB table TestTableForStreams with a simple primary key: Id
Waiting for TestTableForStreams to be created...
Current stream ARN for TestTableForStreams: arn:aws:dynamodb:us-
east-2:123456789012:table/TestTableForStreams/stream/2018-03-20T16:49:55.208
Stream enabled: true
Update view type: NEW_AND_OLD_IMAGES

Performing write activities on TestTableForStreams
Processing item 1 of 100
Processing item 2 of 100
Processing item 3 of 100
...
Processing item 100 of 100
```

```
Shard: {ShardId: shardId-1234567890-...,SequenceNumberRange: {StartingSequenceNumber:
 100002572486797508907,},}
    Shard iterator: EjYFEkX2a26eVTWe...
        StreamRecord(ApproximateCreationDateTime=2025-04-09T13:11:58Z,
 Keys={Id=AttributeValue(S=4)}, NewImage={Message=AttributeValue(S=New Item!),
 Id=AttributeValue(S=4)}, SequenceNumber=200000158404754583390 9, SizeBytes=22,
 StreamViewType=NEW_AND_OLD_IMAGES)
        StreamRecord(ApproximateCreationDateTime=2025-04-09T13:11:58Z,
 Keys={Id=AttributeValue(S=4)}, NewImage={Message=AttributeValue(S=This is an updated
 item), Id=AttributeValue(S=4)}, OldImage={Message=AttributeValue(S=New Item!),
 Id=AttributeValue(S=4)}, SequenceNumber=210000360486976789270 1, SizeBytes=55,
 StreamViewType=NEW_AND_OLD_IMAGES)
        StreamRecord(ApproximateCreationDateTime=2025-04-09T13:11:58Z,
 Keys={Id=AttributeValue(S=4)}, OldImage={Message=AttributeValue(S=This is an updated
 item), Id=AttributeValue(S=4)}, SequenceNumber=220000109977111289843 4, SizeBytes=36,
 StreamViewType=NEW_AND_OLD_IMAGES)
...
Deleting the table...
Table StreamsDemoTable deleted.
Demo complete
```

Example

```
import java.util.HashMap;
import java.util.List;
import java.util.Map;

import software.amazon.awssdk.core.waiters.WaiterResponse;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeAction;
import software.amazon.awssdk.services.dynamodb.model.AttributeDefinition;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.AttributeValueUpdate;
import software.amazon.awssdk.services.dynamodb.model.BillingMode;
import software.amazon.awssdk.services.dynamodb.model.CreateTableRequest;
import software.amazon.awssdk.services.dynamodb.model.CreateTableResponse;
import software.amazon.awssdk.services.dynamodb.model.DeleteItemRequest;
import software.amazon.awssdk.services.dynamodb.model.DeleteTableRequest;
import software.amazon.awssdk.services.dynamodb.model.DescribeStreamRequest;
import software.amazon.awssdk.services.dynamodb.model.DescribeStreamResponse;
import software.amazon.awssdk.services.dynamodb.model.DescribeTableRequest;
import software.amazon.awssdk.services.dynamodb.model.DescribeTableResponse;
```

```
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.GetRecordsRequest;
import software.amazon.awssdk.services.dynamodb.model.GetRecordsResponse;
import software.amazon.awssdk.services.dynamodb.model.GetShardIteratorRequest;
import software.amazon.awssdk.services.dynamodb.model.GetShardIteratorResponse;
import software.amazon.awssdk.services.dynamodb.model.KeySchemaElement;
import software.amazon.awssdk.services.dynamodb.model.KeyType;
import software.amazon.awssdk.services.dynamodb.model.PutItemRequest;
import software.amazon.awssdk.services.dynamodb.model.Record;
import software.amazon.awssdk.services.dynamodb.model.ScalarAttributeType;
import software.amazon.awssdk.services.dynamodb.model.Shard;
import software.amazon.awssdk.services.dynamodb.model.ShardFilter;
import software.amazon.awssdk.services.dynamodb.model.ShardFilterType;
import software.amazon.awssdk.services.dynamodb.model.ShardIteratorType;
import software.amazon.awssdk.services.dynamodb.model.StreamSpecification;
import software.amazon.awssdk.services.dynamodb.model.TableDescription;
import software.amazon.awssdk.services.dynamodb.model.UpdateItemRequest;
import software.amazon.awssdk.services.dynamodb.streams.DynamoDbStreamsClient;
import software.amazon.awssdk.services.dynamodb.waiters.DynamoDbWaiter;

public class StreamsLowLevelDemo {


    public static void main(String[] args) {
        final String usage = "Testing Streams Demo";
        try {
            System.out.println(usage);

            String tableName = "StreamsDemoTable";
            String key = "Id";
            System.out.println("Creating an Amazon DynamoDB table " + tableName + "
 with a simple primary key: " + key);
            Region region = Region.US_WEST_2;
            DynamoDbClient ddb = DynamoDbClient.builder()
                    .region(region)
                    .build();

            DynamoDbStreamsClient ddbStreams = DynamoDbStreamsClient.builder()
                    .region(region)
                    .build();
            DescribeTableRequest describeTableRequest = DescribeTableRequest.builder()
                    .tableName(tableName)
                    .build();
            TableDescription tableDescription = null;
```

```
            try{
                tableDescription = ddb.describeTable(describeTableRequest).table();
            }catch (Exception e){
                System.out.println("Table " + tableName + " does not exist.");
                tableDescription = createTable(ddb, tableName, key);
            }

            // Print the stream settings for the table
            String streamArn = tableDescription.latestStreamArn();

            StreamSpecification streamSpec = tableDescription.streamSpecification();
            System.out.println("Current stream ARN for " + tableDescription.tableName()
 + ": " +
                   streamArn);
            System.out.println("Stream enabled: " + streamSpec.streamEnabled());
            System.out.println("Update view type: " + streamSpec.streamViewType());
            System.out.println();
            // Generate write activity in the table
            System.out.println("Performing write activities on " + tableName);
            int maxItemCount = 100;
            for (Integer i = 1; i <= maxItemCount; i++) {
                System.out.println("Processing item " + i + " of " + maxItemCount);
                // Write a new item
                putItemInTable(key, i, tableName, ddb);
                // Update the item
                updateItemInTable(key, i, tableName, ddb);
                // Delete the item
                deleteDynamoDBItem(key, i, tableName, ddb);
            }

            // Process Stream
            processStream(streamArn, maxItemCount, ddb, ddbStreams, tableName);

            // Delete the table
            System.out.println("Deleting the table...");
            DeleteTableRequest deleteTableRequest = DeleteTableRequest.builder()
                    .tableName(tableName)
                    .build();
            ddb.deleteTable(deleteTableRequest);
            System.out.println("Table " + tableName + " deleted.");
            System.out.println("Demo complete");
            ddb.close();
        } catch (Exception e) {
            System.out.println("Error: " + e.getMessage());
```

```
        }
    }

    private static void processStream(String streamArn, int maxItemCount,
DynamoDbClient ddb, DynamoDbStreamsClient ddbStreams, String tableName) {
        // Get all the shard IDs from the stream. Note that DescribeStream returns
        // the shard IDs one page at a time.
        String lastEvaluatedShardId = null;
        do {
            DescribeStreamRequest describeStreamRequest =
DescribeStreamRequest.builder()
                    .streamArn(streamArn)
                    .exclusiveStartShardId(lastEvaluatedShardId).build();
            DescribeStreamResponse describeStreamResponse =
ddbStreams.describeStream(describeStreamRequest);

            List<Shard> shards = describeStreamResponse.streamDescription().shards();

            // Process each shard on this page

            fetchShardsAndReadRecords(streamArn, maxItemCount, ddbStreams, shards);

            // If LastEvaluatedShardId is set, then there is
            // at least one more page of shard IDs to retrieve
            lastEvaluatedShardId =
describeStreamResponse.streamDescription().lastEvaluatedShardId();

        } while (lastEvaluatedShardId != null);

    }

    private static void fetchShardsAndReadRecords(String streamArn, int maxItemCount,
DynamoDbStreamsClient ddbStreams, List<Shard> shards) {
        for (Shard shard : shards) {
            String shardId = shard.shardId();
            System.out.println("Shard: " + shard);

            // Get an iterator for the current shard
            GetShardIteratorRequest shardIteratorRequest =
GetShardIteratorRequest.builder()
                    .streamArn(streamArn).shardId(shardId)
                    .shardIteratorType(ShardIteratorType.TRIM_HORIZON).build();
```

```
            GetShardIteratorResponse getShardIteratorResult =
ddbStreams.getShardIterator(shardIteratorRequest);

            String currentShardIter = getShardIteratorResult.shardIterator();

            // Shard iterator is not null until the Shard is sealed (marked as
READ_ONLY).
            // To prevent running the loop until the Shard is sealed, we process only
the
            // items that were written into DynamoDB and then exit.
            int processedRecordCount = 0;
            while (currentShardIter != null && processedRecordCount < maxItemCount) {
                // Use the shard iterator to read the stream records
                GetRecordsRequest getRecordsRequest = GetRecordsRequest.builder()
                        .shardIterator(currentShardIter).build();
                GetRecordsResponse getRecordsResult =
ddbStreams.getRecords(getRecordsRequest);
                List<Record> records = getRecordsResult.records();
                for (Record record : records) {
                    System.out.println("        " + record.dynamodb());
                }
                processedRecordCount += records.size();
                currentShardIter = getRecordsResult.nextShardIterator();
            }
            if (currentShardIter == null){
                System.out.println("Shard has been fully processed. Shard iterator is
null.");
                System.out.println("Fetch the child shard to continue processing
instead of bulk fetching all shards");
                DescribeStreamRequest describeStreamRequestForChildShards =
DescribeStreamRequest.builder()
                        .streamArn(streamArn)
                        .shardFilter(ShardFilter.builder()
                                .type(ShardFilterType.CHILD_SHARDS)
                                .shardId(shardId).build())
                        .build();
                DescribeStreamResponse describeStreamResponseChildShards =
ddbStreams.describeStream(describeStreamRequestForChildShards);
                fetchShardsAndReadRecords(streamArn, maxItemCount, ddbStreams,
describeStreamResponseChildShards.streamDescription().shards());
            }
        }
    }
```

```
   private static void putItemInTable(String key, Integer i, String tableName,
DynamoDbClient ddb) {
       Map<String, AttributeValue> item = new HashMap<>();
       item.put(key, AttributeValue.builder()
               .s(i.toString())
               .build());
       item.put("Message", AttributeValue.builder()
               .s("New Item!")
               .build());
       PutItemRequest request = PutItemRequest.builder()
               .tableName(tableName)
               .item(item)
               .build();
       ddb.putItem(request);
   }

   private static void updateItemInTable(String key, Integer i, String tableName,
DynamoDbClient ddb) {

       HashMap<String, AttributeValue> itemKey = new HashMap<>();
       itemKey.put(key, AttributeValue.builder()
               .s(i.toString())
               .build());


       HashMap<String, AttributeValueUpdate> updatedValues = new HashMap<>();
       updatedValues.put("Message", AttributeValueUpdate.builder()
               .value(AttributeValue.builder().s("This is an updated item").build())
               .action(AttributeAction.PUT)
               .build());

       UpdateItemRequest request = UpdateItemRequest.builder()
               .tableName(tableName)
               .key(itemKey)
               .attributeUpdates(updatedValues)
               .build();
       ddb.updateItem(request);
   }

   public static void deleteDynamoDBItem(String key, Integer i, String tableName,
DynamoDbClient ddb) {
       HashMap<String, AttributeValue> keyToGet = new HashMap<>();
       keyToGet.put(key, AttributeValue.builder()
               .s(i.toString())
```

```
                .build());

        DeleteItemRequest deleteReq = DeleteItemRequest.builder()
                .tableName(tableName)
                .key(keyToGet)
                .build();
        ddb.deleteItem(deleteReq);
    }

    public static TableDescription createTable(DynamoDbClient ddb, String tableName,
String key) {
        DynamoDbWaiter dbWaiter = ddb.waiter();
        StreamSpecification streamSpecification = StreamSpecification.builder()
                .streamEnabled(true)
                .streamViewType("NEW_AND_OLD_IMAGES")
                .build();
        CreateTableRequest request = CreateTableRequest.builder()
                .attributeDefinitions(AttributeDefinition.builder()
                        .attributeName(key)
                        .attributeType(ScalarAttributeType.S)
                        .build())
                .keySchema(KeySchemaElement.builder()
                        .attributeName(key)
                        .keyType(KeyType.HASH)
                        .build())
                .billingMode(BillingMode.PAY_PER_REQUEST) //  DynamoDB automatically
scales based on traffic.
                .tableName(tableName)
                .streamSpecification(streamSpecification)
                .build();

        TableDescription newTable;
        try {
            CreateTableResponse response = ddb.createTable(request);
            DescribeTableRequest tableRequest = DescribeTableRequest.builder()
                    .tableName(tableName)
                    .build();

            System.out.println("Waiting for " + tableName + " to be created...");

            // Wait until the Amazon DynamoDB table is created.
            WaiterResponse<DescribeTableResponse> waiterResponse =
dbWaiter.waitUntilTableExists(tableRequest);
            waiterResponse.matched().response().ifPresent(System.out::println);
```

```
            newTable = response.tableDescription();
            return newTable;

        } catch (DynamoDbException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
        return null;
    }



}
```

DynamoDB Streams and AWS Lambda triggers

Amazon DynamoDB is integrated with AWS Lambda so that you can create *triggers*—pieces of code that automatically respond to events in DynamoDB Streams. With triggers, you can build applications that react to data modifications in DynamoDB tables.

Topics

- Tutorial #1: Using filters to process all events with Amazon DynamoDB and AWS Lambda using the AWS CLI
- Tutorial #2: Using filters to process some events with DynamoDB and Lambda
- Best practices using DynamoDB Streams with Lambda

If you enable DynamoDB Streams on a table, you can associate the stream Amazon Resource Name (ARN) with an AWS Lambda function that you write. All mutation actions to that DynamoDB table can then be captured as an item on the stream. For example, you can set a trigger so that when an item in a table is modified a new record immediately appears in that table's stream.

> **Note**
>
> If you subscribe more than two Lambda functions to one DynamoDB stream, read throttling might occur.

The AWS Lambda service polls the stream for new records four times per second. When new stream records are available, your Lambda function is synchronously invoked. You can subscribe up

to two Lambda functions to the same DynamoDB stream. If you subscribe more than two Lambda functions to the same DynamoDB stream, read throttling might occur.

The Lambda function can send a notification, initiate a workflow, or perform many other actions that you specify. You can write a Lambda function to simply copy each stream record to persistent storage, such as Amazon S3 File Gateway (Amazon S3), and create a permanent audit trail of write activity in your table. Or suppose that you have a mobile gaming app that writes to a `GameScores` table. Whenever the `TopScore` attribute of the `GameScores` table is updated, a corresponding stream record is written to the table's stream. This event could then trigger a Lambda function that posts a congratulatory message on a social media network. This function could also be written to ignore any stream records that are not updates to `GameScores`, or that do not modify the `TopScore` attribute.

If your function returns an error, Lambda retries the batch until it processes successfully or the data expires. You can also configure Lambda to retry with a smaller batch, limit the number of retries, discard records once they become too old, and other options.

As performance best practices, the Lambda function needs to be short lived. To avoid introducing unnecessary processing delays, it also should not execute complex logic. For a high velocity stream in particular, it is better to trigger an asynchronous post-processing step function workflows than synchronous long running Lambdas.

You can use Lambda triggers across different AWS accounts by configuring a resource-based policy on the DynamoDB stream to grant the cross-account read access to Lambda function. To learn more about how to configure your stream to allow cross-account access, see Share access with cross-account AWS Lambda functions in the DynamoDB Developer Guide.

For more information about AWS Lambda, see the AWS Lambda Developer Guide.

Tutorial #1: Using filters to process all events with Amazon DynamoDB and AWS Lambda using the AWS CLI

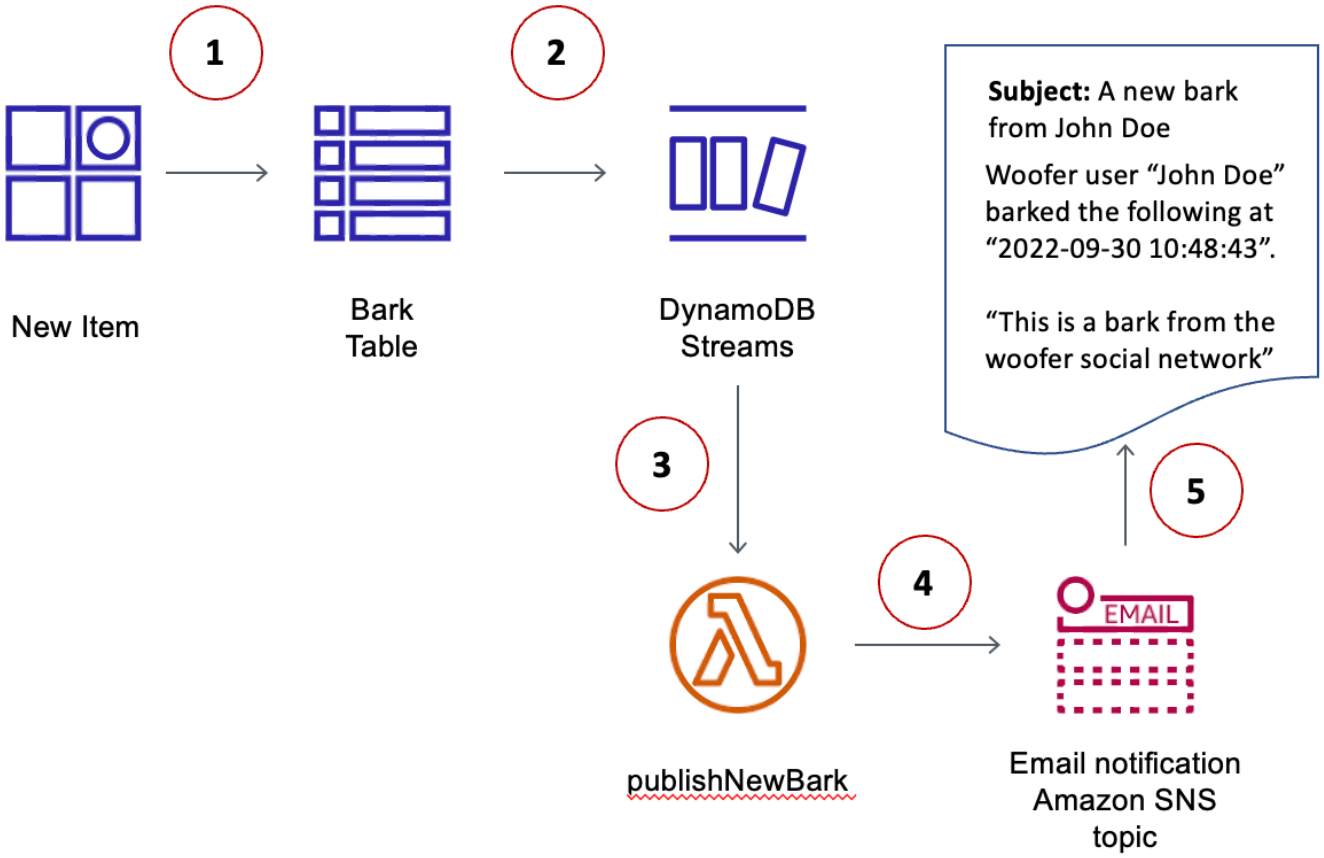
In this tutorial, you will create an AWS Lambda trigger to process a stream from a DynamoDB table.

Topics

- Step 1: Create a DynamoDB table with a stream enabled
- Step 2: Create a Lambda execution role
- Step 3: Create an Amazon SNS topic

- Step 4: Create and test a Lambda function
- Step 5: Create and test a trigger

The scenario for this tutorial is Woofer, a simple social network. Woofer users communicate using *barks* (short text messages) that are sent to other Woofer users. The following diagram shows the components and workflow for this application.



1. A user writes an item to a DynamoDB table (`BarkTable`). Each item in the table represents a bark.
2. A new stream record is written to reflect that a new item has been added to `BarkTable`.
3. The new stream record triggers an AWS Lambda function (`publishNewBark`).
4. If the stream record indicates that a new item was added to `BarkTable`, the Lambda function reads the data from the stream record and publishes a message to a topic in Amazon Simple Notification Service (Amazon SNS).
5. The message is received by subscribers to the Amazon SNS topic. (In this tutorial, the only subscriber is an email address.)

Before You Begin

This tutorial uses the AWS Command Line Interface AWS CLI. If you have not done so already, follow the instructions in the AWS Command Line Interface User Guide to install and configure the AWS CLI.

Step 1: Create a DynamoDB table with a stream enabled

In this step, you create a DynamoDB table (`BarkTable`) to store all of the barks from Woofer users. The primary key is composed of `Username` (partition key) and `Timestamp` (sort key). Both of these attributes are of type string.

`BarkTable` has a stream enabled. Later in this tutorial, you create a trigger by associating an AWS Lambda function with the stream.

1. Enter the following command to create the table.

```
aws dynamodb create-table \
    --table-name BarkTable \
    --attribute-definitions AttributeName=Username,AttributeType=S
 AttributeName=Timestamp,AttributeType=S \
    --key-schema AttributeName=Username,KeyType=HASH
 AttributeName=Timestamp,KeyType=RANGE \
    --provisioned-throughput ReadCapacityUnits=5,WriteCapacityUnits=5 \
    --stream-specification StreamEnabled=true,StreamViewType=NEW_AND_OLD_IMAGES
```

2. In the output, look for the `LatestStreamArn`.

```
...
"LatestStreamArn": "arn:aws:dynamodb:region:accountID:table/BarkTable/
stream/timestamp
...
```

   Make a note of the *`region`* and the *`accountID`*, because you need them for the other steps in this tutorial.

Step 2: Create a Lambda execution role

In this step, you create an AWS Identity and Access Management (IAM) role (`WooferLambdaRole`) and assign permissions to it. This role is used by the Lambda function that you create in Step 4: Create and test a Lambda function.

You also create a policy for the role. The policy contains all of the permissions that the Lambda function needs at runtime.

1. Create a file named `trust-relationship.json` with the following contents.

   JSON

```
{
   "Version":"2012-10-17",
   "Statement": [
     {
       "Effect": "Allow",
       "Principal": {
         "Service": "lambda.amazonaws.com"
       },
       "Action": "sts:AssumeRole"
     }
   ]
 }
```

2. Enter the following command to create `WooferLambdaRole`.

```
aws iam create-role --role-name WooferLambdaRole \
    --path "/service-role/" \
    --assume-role-policy-document file://trust-relationship.json
```

3. Create a file named `role-policy.json` with the following contents. (Replace *`region`* and *`accountID`* with your AWS Region and account ID.)

   JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "logs:CreateLogGroup",
                "logs:CreateLogStream",
                "logs:PutLogEvents"
            ],
```

```
            "Resource": "arn:aws:logs:us-east-1:111122223333:*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "dynamodb:DescribeStream",
                "dynamodb:GetRecords",
                "dynamodb:GetShardIterator",
                "dynamodb:ListStreams"
            ],
            "Resource": "arn:aws:dynamodb:us-east-1:111122223333:table/
BarkTable/stream/*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "sns:Publish"
            ],
            "Resource": [
                "*"
            ]
        }
    ]
}
```

The policy has four statements that allow `WooferLambdaRole` to do the following:

- Run a Lambda function (`publishNewBark`). You create the function later in this tutorial.
- Access Amazon CloudWatch Logs. The Lambda function writes diagnostics to CloudWatch Logs at runtime.
- Read data from the DynamoDB stream for `BarkTable`.
- Publish messages to Amazon SNS.

4. Enter the following command to attach the policy to `WooferLambdaRole`.

```
aws iam put-role-policy --role-name WooferLambdaRole \
    --policy-name WooferLambdaRolePolicy \
    --policy-document file://role-policy.json
```

Step 3: Create an Amazon SNS topic

In this step, you create an Amazon SNS topic (`wooferTopic`) and subscribe an email address to it. Your Lambda function uses this topic to publish new barks from Woofer users.

1. Enter the following command to create a new Amazon SNS topic.

```
aws sns create-topic --name wooferTopic
```

2. Enter the following command to subscribe an email address to `wooferTopic`. (Replace *region* and *accountID* with your AWS Region and account ID, and replace *example@example.com* with a valid email address.)

```
aws sns subscribe \
    --topic-arn arn:aws:sns:region:accountID:wooferTopic \
    --protocol email \
    --notification-endpoint example@example.com
```

3. Amazon SNS sends a confirmation message to your email address. Choose the **Confirm subscription** link in that message to complete the subscription process.

Step 4: Create and test a Lambda function

In this step, you create an AWS Lambda function (`publishNewBark`) to process stream records from `BarkTable`.

The `publishNewBark` function processes only the stream events that correspond to new items in `BarkTable`. The function reads data from such an event, and then invokes Amazon SNS to publish it.

1. Create a file named `publishNewBark.js` with the following contents. Replace *region* and *accountID* with your AWS Region and account ID.

```
'use strict';
var AWS = require("aws-sdk");
var sns = new AWS.SNS();

exports.handler = (event, context, callback) => {

    event.Records.forEach((record) => {
        console.log('Stream record: ', JSON.stringify(record, null, 2));
```

```
        if (record.eventName == 'INSERT') {
            var who = JSON.stringify(record.dynamodb.NewImage.Username.S);
            var when = JSON.stringify(record.dynamodb.NewImage.Timestamp.S);
            var what = JSON.stringify(record.dynamodb.NewImage.Message.S);
            var params = {
                Subject: 'A new bark from ' + who,
                Message: 'Woofer user ' + who + ' barked the following at ' + when
 + ':\n\n ' + what,
                TopicArn: 'arn:aws:sns:region:accountID:wooferTopic'
            };
            sns.publish(params, function(err, data) {
                if (err) {
                    console.error("Unable to send message. Error JSON:",
 JSON.stringify(err, null, 2));
                } else {
                    console.log("Results from sending message: ",
 JSON.stringify(data, null, 2));
                }
            });
        }
    });
    callback(null, `Successfully processed ${event.Records.length} records.`);
};
```

2. Create a zip file to contain `publishNewBark.js`. If you have the zip command-line utility, you can enter the following command to do this.

```
zip publishNewBark.zip publishNewBark.js
```

3. When you create the Lambda function, you specify the Amazon Resource Name (ARN) for `WooferLambdaRole`, which you created in Step 2: Create a Lambda execution role. Enter the following command to retrieve this ARN.

```
aws iam get-role --role-name WooferLambdaRole
```

   In the output, look for the ARN for `WooferLambdaRole`.

```
...
"Arn": "arn:aws:iam::region:role/service-role/WooferLambdaRole"
...
```

Enter the following command to create the Lambda function. Replace *roleARN* with the ARN for `WooferLambdaRole`.

```
aws lambda create-function \
    --region region \
    --function-name publishNewBark \
    --zip-file fileb://publishNewBark.zip \
    --role roleARN \
    --handler publishNewBark.handler \
    --timeout 5 \
    --runtime nodejs16.x
```

4. Now test `publishNewBark` to verify that it works. To do this, you provide input that resembles a real record from DynamoDB Streams.

   Create a file named `payload.json` with the following contents. Replace *region* and *accountID* with your AWS Region and account ID.

```
{
    "Records": [
        {
            "eventID": "7de3041dd709b024af6f29e4fa13d34c",
            "eventName": "INSERT",
            "eventVersion": "1.1",
            "eventSource": "aws:dynamodb",
            "awsRegion": "region",
            "dynamodb": {
                "ApproximateCreationDateTime": 1479499740,
                "Keys": {
                    "Timestamp": {
                        "S": "2016-11-18:12:09:36"
                    },
                    "Username": {
                        "S": "John Doe"
                    }
                },
                "NewImage": {
                    "Timestamp": {
                        "S": "2016-11-18:12:09:36"
                    },
                    "Message": {
```

```
                        "S": "This is a bark from the Woofer social network"
                    },
                    "Username": {
                        "S": "John Doe"
                    }
                },
                "SequenceNumber": "130216000000000001596893679",
                "SizeBytes": 112,
                "StreamViewType": "NEW_IMAGE"
            },
            "eventSourceARN": "arn:aws:dynamodb:region:account ID:table/BarkTable/
stream/2016-11-16T20:42:48.104"
        }
    ]
}
```

Enter the following command to test the `publishNewBark` function.

```
aws lambda invoke --function-name publishNewBark --payload file://payload.json --
cli-binary-format raw-in-base64-out output.txt
```

If the test was successful, you will see the following output.

```
{
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
}
```

In addition, the `output.txt` file will contain the following text.

```
"Successfully processed 1 records."
```

You will also receive a new email message within a few minutes.

> **Note**
>
> AWS Lambda writes diagnostic information to Amazon CloudWatch Logs. If you encounter errors with your Lambda function, you can use these diagnostics for troubleshooting purposes:

1. Open the CloudWatch console at https://console.aws.amazon.com/cloudwatch/.
2. In the navigation pane, choose **Logs**.
3. Choose the following log group: `/aws/lambda/publishNewBark`
4. Choose the latest log stream to view the output (and errors) from the function.

Step 5: Create and test a trigger

In Step 4: Create and test a Lambda function, you tested the Lambda function to ensure that it ran correctly. In this step, you create a *trigger* by associating the Lambda function (`publishNewBark`) with an event source (the `BarkTable` stream).

1. When you create the trigger, you need to specify the ARN for the `BarkTable` stream. Enter the following command to retrieve this ARN.

   ```
   aws dynamodb describe-table --table-name BarkTable
   ```

   In the output, look for the `LatestStreamArn`.

   ```
   ...
    "LatestStreamArn": "arn:aws:dynamodb:region:accountID:table/BarkTable/
   stream/timestamp
   ...
   ```

2. Enter the following command to create the trigger. Replace *`streamARN`* with the actual stream ARN.

   ```
   aws lambda create-event-source-mapping \
       --region region \
       --function-name publishNewBark \
       --event-source streamARN  \
       --batch-size 1 \
       --starting-position TRIM_HORIZON
   ```

3. Test the trigger. Enter the following command to add an item to `BarkTable`.

   ```
   aws dynamodb put-item \
       --table-name BarkTable \
   ```

```
    --item Username={S="Jane
Doe"},Timestamp={S="2016-11-18:14:32:17"},Message={S="Testing...1...2...3"}
```

You should receive a new email message within a few minutes.

4. Open the DynamoDB console and add a few more items to `BarkTable`. You must specify values for the `Username` and `Timestamp` attributes. (You should also specify a value for `Message`, even though it is not required.) You should receive a new email message for each item you add to `BarkTable`.

   The Lambda function processes only new items that you add to `BarkTable`. If you update or delete an item in the table, the function does nothing.

> **Note**
>
> AWS Lambda writes diagnostic information to Amazon CloudWatch Logs. If you encounter errors with your Lambda function, you can use these diagnostics for troubleshooting purposes.
>
> 1. Open the CloudWatch console at https://console.aws.amazon.com/cloudwatch/.
> 2. In the navigation pane, choose **Logs**.
> 3. Choose the following log group: `/aws/lambda/publishNewBark`
> 4. Choose the latest log stream to view the output (and errors) from the function.

Tutorial #2: Using filters to process some events with DynamoDB and Lambda

In this tutorial, you will create an AWS Lambda trigger to process only some events in a stream from a DynamoDB table.

Topics

- Putting it all together - CloudFormation
- Putting it all together - CDK

With Lambda event filtering you can use filter expressions to control which events Lambda sends to your function for processing. You can configure up to 5 different filters per DynamoDB streams.

If you are using batching windows, Lambda applies the filter criteria to each new event to see if it should be included in the current batch.

Filters are applied via structures called `FilterCriteria`. The 3 main attributes of `FilterCriteria` are `metadata properties`, `data properties` and `filter patterns`.

Here is an example structure of a DynamoDB Streams event:

```
{
  "eventID": "c9fbe7d0261a5163fcb6940593e41797",
  "eventName": "INSERT",
  "eventVersion": "1.1",
  "eventSource": "aws:dynamodb",
  "awsRegion": "us-east-2",
  "dynamodb": {
    "ApproximateCreationDateTime": 1664559083.0,
    "Keys": {
      "SK": { "S": "PRODUCT#CHOCOLATE#DARK#1000" },
      "PK": { "S": "COMPANY#1000" }
    },
    "NewImage": {
      "quantity": { "N": "50" },
      "company_id": { "S": "1000" },
      "fabric": { "S": "Florida Chocolates" },
      "price": { "N": "15" },
      "stores": { "N": "5" },
      "product_id": { "S": "1000" },
      "SK": { "S": "PRODUCT#CHOCOLATE#DARK#1000" },
      "PK": { "S": "COMPANY#1000" },
      "state": { "S": "FL" },
      "type": { "S": "" }
    },
    "SequenceNumber": "700000000000888747038",
    "SizeBytes": 174,
    "StreamViewType": "NEW_AND_OLD_IMAGES"
  },
  "eventSourceARN": "arn:aws:dynamodb:us-east-2:111122223333:table/chocolate-table-
StreamsSampleDDBTable-LUOI6UXQY7J1/stream/2022-09-30T17:05:53.209"
}
```

The `metadata properties` are the fields of the event object. In the case of DynamoDB Streams, the `metadata properties` are fields like dynamodb or `eventName`.

The `data properties` are the fields of the event body. To filter on `data properties`, make sure to contain them in `FilterCriteria` within the proper key. For DynamoDB event sources, the data key is `NewImage` or `OldImage`.

Finally, the filter rules will define the filter expression that you want to apply to a specific property. Here are some examples:

Comparison operator	Example	Rule syntax (Partial)
Null	Product Type is null	`{ "product_type": { "S": null } }`
Empty	Product name is empty	`{ "product_name": { "S": [ ""] } }`
Equals	State equals Florida	`{ "state": { "S": ["FL"] } }`
And	Product state equals Florida and product category Chocolate	`{ "state": { "S": ["FL"] } , "category": { "S": [ "CHOCOLATE"] } }`
Or	Product state is Florida or California	`{ "state": { "S": ["FL","CA"] } }`
Not	Product state is not Florida	`{"state": {"S": [{"anything-but": ["FL"]}]}}`
Exists	Product Homemade exists	`{"homemade": {"S": [{"exists": true}]}}`
Does not exist	Product Homemade does not exist	`{"homemade": {"S": [{"exists": false}]}}`
Begins with	PK begins with COMPANY	`{"PK": {"S": [{"prefix": "COMPANY"}]}}`

You can specify up to 5 event filtering patterns for a Lambda function. Notice that each one of those 5 events will be evaluated as a logical OR. So if you configure two filters named `Filter_One` and `Filter_Two`, the Lambda function will execute `Filter_One` OR `Filter_Two`.

> **Note**
>
> In the Lambda event filtering page there are some options to filter and compare numeric values, however in the case of DynamoDB filter events it doesn't apply because numbers in DynamoDB are stored as strings. For example `"quantity": { "N": "50" }`, we know its a number because of the `"N"` property.

Putting it all together - CloudFormation

To show event filtering functionality in practice, here is a sample CloudFormation template. This template will generate a Simple DynamoDB table with a Partition Key PK and a Sort Key SK with Amazon DynamoDB Streams enabled. It will create a lambda function and a simple Lambda Execution role that will allow write logs to Amazon Cloudwatch, and read the events from the Amazon DynamoDB Stream. It will also add the event source mapping between the DynamoDB Streams and the Lambda function, so the function can be executed every time there is an event in the Amazon DynamoDB Stream.

```
AWSTemplateFormatVersion: "2010-09-09"

Description: Sample application that presents AWS Lambda event source filtering
with Amazon DynamoDB Streams.

Resources:
  StreamsSampleDDBTable:
    Type: AWS::DynamoDB::Table
    Properties:
      AttributeDefinitions:
        - AttributeName: "PK"
          AttributeType: "S"
        - AttributeName: "SK"
          AttributeType: "S"
      KeySchema:
        - AttributeName: "PK"
          KeyType: "HASH"
        - AttributeName: "SK"
          KeyType: "RANGE"
```

```
      StreamSpecification:
        StreamViewType: "NEW_AND_OLD_IMAGES"
      ProvisionedThroughput:
        ReadCapacityUnits: 5
        WriteCapacityUnits: 5

  LambdaExecutionRole:
    Type: AWS::IAM::Role
    Properties:
      AssumeRolePolicyDocument:
        Version: "2012-10-17",
        Statement:
          - Effect: Allow
            Principal:
              Service:
                - lambda.amazonaws.com
            Action:
              - sts:AssumeRole
      Path: "/"
      Policies:
        - PolicyName: root
          PolicyDocument:
            Version: "2012-10-17",
            Statement:
              - Effect: Allow
                Action:
                  - logs:CreateLogGroup
                  - logs:CreateLogStream
                  - logs:PutLogEvents
                Resource: arn:aws:logs:*:*:*
              - Effect: Allow
                Action:
                  - dynamodb:DescribeStream
                  - dynamodb:GetRecords
                  - dynamodb:GetShardIterator
                  - dynamodb:ListStreams
                Resource: !GetAtt StreamsSampleDDBTable.StreamArn

  EventSourceDDBTableStream:
    Type: AWS::Lambda::EventSourceMapping
    Properties:
      BatchSize: 1
      Enabled: True
      EventSourceArn: !GetAtt StreamsSampleDDBTable.StreamArn
```

```
      FunctionName: !GetAtt ProcessEventLambda.Arn
      StartingPosition: LATEST

  ProcessEventLambda:
    Type: AWS::Lambda::Function
    Properties:
      Runtime: python3.7
      Timeout: 300
      Handler: index.handler
      Role: !GetAtt LambdaExecutionRole.Arn
      Code:
        ZipFile: |
          import logging

          LOGGER = logging.getLogger()
          LOGGER.setLevel(logging.INFO)

          def handler(event, context):
            LOGGER.info('Received Event: %s', event)
            for rec in event['Records']:
              LOGGER.info('Record: %s', rec)

Outputs:
  StreamsSampleDDBTable:
    Description: DynamoDB Table ARN created for this example
    Value: !GetAtt StreamsSampleDDBTable.Arn
  StreamARN:
    Description: DynamoDB Table ARN created for this example
    Value: !GetAtt StreamsSampleDDBTable.StreamArn
```

After you deploy this cloud formation template you can insert the following Amazon DynamoDB Item:

```
{
 "PK": "COMPANY#1000",
 "SK": "PRODUCT#CHOCOLATE#DARK",
 "company_id": "1000",
 "type": "",
 "state": "FL",
 "stores": 5,
 "price": 15,
 "quantity": 50,
 "fabric": "Florida Chocolates"
```

```
}
```

Thanks to the simple lambda function included inline in this cloud formation template, you will see the events in the Amazon CloudWatch log groups for the lambda function as follows:

```
{
  "eventID": "c9fbe7d0261a5163fcb6940593e41797",
  "eventName": "INSERT",
  "eventVersion": "1.1",
  "eventSource": "aws:dynamodb",
  "awsRegion": "us-east-2",
  "dynamodb": {
    "ApproximateCreationDateTime": 1664559083.0,
    "Keys": {
      "SK": { "S": "PRODUCT#CHOCOLATE#DARK#1000" },
      "PK": { "S": "COMPANY#1000" }
    },
    "NewImage": {
      "quantity": { "N": "50" },
      "company_id": { "S": "1000" },
      "fabric": { "S": "Florida Chocolates" },
      "price": { "N": "15" },
      "stores": { "N": "5" },
      "product_id": { "S": "1000" },
      "SK": { "S": "PRODUCT#CHOCOLATE#DARK#1000" },
      "PK": { "S": "COMPANY#1000" },
      "state": { "S": "FL" },
      "type": { "S": "" }
    },
    "SequenceNumber": "700000000000888747038",
    "SizeBytes": 174,
    "StreamViewType": "NEW_AND_OLD_IMAGES"
  },
  "eventSourceARN": "arn:aws:dynamodb:us-east-2:111122223333:table/chocolate-table-
StreamsSampleDDBTable-LUOI6UXQY7J1/stream/2022-09-30T17:05:53.209"
}
```

Filter Examples

- **Only products that matches a given state**

This example modifies the CloudFormation template to include a filter to match all products which come from Florida, with the abbreviation “FL”.

```
EventSourceDDBTableStream:
    Type: AWS::Lambda::EventSourceMapping
    Properties:
      BatchSize: 1
      Enabled: True
      FilterCriteria:
        Filters:
          - Pattern: '{ "dynamodb": { "NewImage": { "state": { "S": ["FL"] } } } }'
      EventSourceArn: !GetAtt StreamsSampleDDBTable.StreamArn
      FunctionName: !GetAtt ProcessEventLambda.Arn
      StartingPosition: LATEST
```

Once you redeploy the stack, you can add the following DynamoDB item to the table. Note that it will not appear in the Lambda function logs, because the product in this example is from California.

```
{
 "PK": "COMPANY#1000",
 "SK": "PRODUCT#CHOCOLATE#DARK#1000",
 "company_id": "1000",
 "fabric": "Florida Chocolates",
 "price": 15,
 "product_id": "1000",
 "quantity": 50,
 "state": "CA",
 "stores": 5,
 "type": ""
}
```

- **Only the items that starts with some values in the PK and SK**

This example modifies the CloudFormation template to include the following condition:

```
EventSourceDDBTableStream:
    Type: AWS::Lambda::EventSourceMapping
    Properties:
      BatchSize: 1
      Enabled: True
      FilterCriteria:
```

```
        Filters:
          - Pattern: '{"dynamodb": {"Keys": {"PK": { "S": [{ "prefix":
"COMPANY" }] },"SK": { "S": [{ "prefix": "PRODUCT" }] }}}}'
      EventSourceArn: !GetAtt StreamsSampleDDBTable.StreamArn
      FunctionName: !GetAtt ProcessEventLambda.Arn
      StartingPosition: LATEST
```

Notice the AND condition requires the condition to be inside the pattern, where Keys PK and SK are in the same expression separated by comma.

Either start with some values on PK and SK or is from certain state.

This example modifies the CloudFormation template to include the following conditions:

```
 EventSourceDDBTableStream:
   Type: AWS::Lambda::EventSourceMapping
   Properties:
     BatchSize: 1
     Enabled: True
     FilterCriteria:
       Filters:
         - Pattern: '{"dynamodb": {"Keys": {"PK": { "S": [{ "prefix":
"COMPANY" }] },"SK": { "S": [{ "prefix": "PRODUCT" }] }}}}'
         - Pattern: '{ "dynamodb": { "NewImage": { "state": { "S": ["FL"] } } } }'
     EventSourceArn: !GetAtt StreamsSampleDDBTable.StreamArn
     FunctionName: !GetAtt ProcessEventLambda.Arn
     StartingPosition: LATEST
```

Notice the OR condition is added by introducing new patterns in the filter section.

Putting it all together - CDK

The following sample CDK project formation template walks through event filtering functionality. Before working with this CDK project you will need to install the pre-requisites including running preparation scripts.

Create a CDK project

First create a new AWS CDK project, by invoking `cdk init` in an empty directory.

```
mkdir ddb_filters
```

```
cd ddb_filters
cdk init app --language python
```

The `cdk init` command uses the name of the project folder to name various elements of the project, including classes, subfolders, and files. Any hyphens in the folder name are converted to underscores. The name should otherwise follow the form of a Python identifier. For example, it should not start with a number or contain spaces.

To work with the new project, activate its virtual environment. This allows the project's dependencies to be installed locally in the project folder, instead of globally.

```
source .venv/bin/activate
python -m pip install -r requirements.txt
```

> **Note**
>
> You may recognize this as the Mac/Linux command to activate a virtual environment. The Python templates include a batch file, `source.bat`, that allows the same command to be used on Windows. The traditional Windows command `.venv\Scripts\activate.bat` works too. If you initialized your AWS CDK project using AWS CDK Toolkit v1.70.0 or earlier, your virtual environment is in the `.env` directory instead of `.venv`.

Base Infrastructure

Open the file `./ddb_filters/ddb_filters_stack.py` with your preferred text editor. This file was auto generated when you created the AWS CDK project.

Next, add the functions `_create_ddb_table` and `_set_ddb_trigger_function`. These functions will create a DynamoDB table with partition key PK and sort key SK in provision mode on-demand mode, with Amazon DynamoDB Streams enabled by default to show New and Old images.

The Lambda function will be stored in the folder `lambda` under the file `app.py`. This file will be created later. It will include an environment variable `APP_TABLE_NAME`, which will be the name of the Amazon DynamoDB Table created by this stack. In the same function we will grant stream read permissions to the Lambda function. Finally, it will subscribe to the DynamoDB Streams as the event source for the lambda function.

At the end of the file in the `__init__` method, you will call the respective constructs to initialize them in the stack. For bigger projects that require additional components and services, it might be best to define these constructs outside the base stack.

```
import os
import json

import aws_cdk as cdk
from aws_cdk import (
    Stack,
    aws_lambda as _lambda,
    aws_dynamodb as dynamodb,
)
from constructs import Construct


class DdbFiltersStack(Stack):

    def _create_ddb_table(self):
        dynamodb_table = dynamodb.Table(
            self,
            "AppTable",
            partition_key=dynamodb.Attribute(
                name="PK", type=dynamodb.AttributeType.STRING
            ),
            sort_key=dynamodb.Attribute(
                name="SK", type=dynamodb.AttributeType.STRING),
            billing_mode=dynamodb.BillingMode.PAY_PER_REQUEST,
            stream=dynamodb.StreamViewType.NEW_AND_OLD_IMAGES,
            removal_policy=cdk.RemovalPolicy.DESTROY,
        )

        cdk.CfnOutput(self, "AppTableName", value=dynamodb_table.table_name)
        return dynamodb_table

    def _set_ddb_trigger_function(self, ddb_table):
        events_lambda = _lambda.Function(
            self,
            "LambdaHandler",
            runtime=_lambda.Runtime.PYTHON_3_9,
            code=_lambda.Code.from_asset("lambda"),
            handler="app.handler",
            environment={
```

```
                "APP_TABLE_NAME": ddb_table.table_name,
            },
        )

        ddb_table.grant_stream_read(events_lambda)

        event_subscription = _lambda.CfnEventSourceMapping(
            scope=self,
            id="companyInsertsOnlyEventSourceMapping",
            function_name=events_lambda.function_name,
            event_source_arn=ddb_table.table_stream_arn,
            maximum_batching_window_in_seconds=1,
            starting_position="LATEST",
            batch_size=1,
        )

    def __init__(self, scope: Construct, construct_id: str, **kwargs) -> None:
        super().__init__(scope, construct_id, **kwargs)

        ddb_table = self._create_ddb_table()
        self._set_ddb_trigger_function(ddb_table)
```

Now we will create a very simple lambda function that will print the logs into Amazon CloudWatch. To do this, create a new folder called `lambda`.

```
mkdir lambda
touch app.py
```

Using your favorite text editor, add the following content to the `app.py` file:

```
import logging

LOGGER = logging.getLogger()
LOGGER.setLevel(logging.INFO)


def handler(event, context):
    LOGGER.info('Received Event: %s', event)
    for rec in event['Records']:
        LOGGER.info('Record: %s', rec)
```

Ensuring you are in the `/ddb_filters/` folder, type the following command to create the sample application:

```
cdk deploy
```

At some point you will be asked to confirm if you want to deploy the solution. Accept the changes by typing Y.

```
##########################################################################################
# + # ${LambdaHandler/ServiceRole} # arn:${AWS::Partition}:iam::aws:policy/service-
role/AWSLambdaBasicExecutionRole #
##########################################################################################

Do you wish to deploy these changes (y/n)? y

...

#  Deployment time: 67.73s

Outputs:
DdbFiltersStack.AppTableName = DdbFiltersStack-AppTable815C50BC-1M1W7209V5YPP
Stack ARN:
arn:aws:cloudformation:us-east-2:111122223333:stack/
DdbFiltersStack/66873140-40f3-11ed-8e93-0a74f296a8f6
```

Once the changes are deployed, open your AWS console and add one item to your table.

```
{
 "PK": "COMPANY#1000",
 "SK": "PRODUCT#CHOCOLATE#DARK",
 "company_id": "1000",
 "type": "",
 "state": "FL",
 "stores": 5,
 "price": 15,
 "quantity": 50,
 "fabric": "Florida Chocolates"
}
```

The CloudWatch logs should now contain all the information from this entry.

Filter Examples

- **Only products that matches a given state**

Open the file `ddb_filters/ddb_filters/ddb_filters_stack.py`, and modify it to include the filter that matches all the products that are equals to “FL”. This can be revised just below the `event_subscription` in line 45.

```
event_subscription.add_property_override(
    property_path="FilterCriteria",
    value={
        "Filters": [
            {
                "Pattern": json.dumps(
                    {"dynamodb": {"NewImage": {"state": {"S": ["FL"]}}}}
                )
            },
        ]
    },
)
```

- **Only the items that starts with some values in the PK and SK**

Modify the python script to include the following condition:

```
event_subscription.add_property_override(
    property_path="FilterCriteria",
    value={
        "Filters": [
            {
                "Pattern": json.dumps(
                    {
                        {
                            "dynamodb": {
                                "Keys": {
                                    "PK": {"S": [{"prefix": "COMPANY"}]},
                                    "SK": {"S": [{"prefix": "PRODUCT"}]},
                                }
                            }
                        }
                    }
                )
            },
```

```
        ]
    },
```

- **Either start with some values on PK and SK or is from certain state.**

Modify the python script to include the following conditions:

```
event_subscription.add_property_override(
    property_path="FilterCriteria",
    value={
        "Filters": [
            {
                "Pattern": json.dumps(
                    {
                        {
                            "dynamodb": {
                                "Keys": {
                                    "PK": {"S": [{"prefix": "COMPANY"}]},
                                    "SK": {"S": [{"prefix": "PRODUCT"}]},
                                }
                            }
                        }
                    }
                )
            },
            {
                "Pattern": json.dumps(
                    {"dynamodb": {"NewImage": {"state": {"S": ["FL"]}}}}
                )
            },
        ]
    },
)
```

Notice that the OR condition is added by adding more elements to the Filters array.

Cleanup

Locate the filter stack in the base of your working directory, and execute `cdk destroy`. You will be asked to confirm the resource deletion:

```
cdk destroy
```

```
Are you sure you want to delete: DdbFiltersStack (y/n)? y
```

Best practices using DynamoDB Streams with Lambda

An AWS Lambda function runs within a *container*—an execution environment that is isolated from other functions. When you run a function for the first time, AWS Lambda creates a new container and begins executing the function's code.

A Lambda function has a *handler* that is run once per invocation. The handler contains the main business logic for the function. For example, the Lambda function shown in Step 4: Create and test a Lambda function has a handler that can process records in a DynamoDB stream.

You can also provide initialization code that runs one time only—after the container is created, but before AWS Lambda runs the handler for the first time. The Lambda function shown in Step 4: Create and test a Lambda function has initialization code that imports the SDK for JavaScript in Node.js, and creates a client for Amazon SNS. These objects should only be defined once, outside of the handler.

After the function runs, AWS Lambda might opt to reuse the container for subsequent invocations of the function. In this case, your function handler might be able to reuse the resources that you defined in your initialization code. (You cannot control how long AWS Lambda will retain the container, or whether the container will be reused at all.)

For DynamoDB triggers using AWS Lambda, we recommend the following:

- AWS service clients should be instantiated in the initialization code, not in the handler. This allows AWS Lambda to reuse existing connections, for the duration of the container's lifetime.
- In general, you do not need to explicitly manage connections or implement connection pooling because AWS Lambda manages this for you.

A Lambda consumer for a DynamoDB stream doesn't guarantee exactly once delivery and may lead to occasional duplicates. Make sure your Lambda function code is idempotent to prevent unexpected issues from arising because of duplicate processing.

For more information, see Best practices for working with AWS Lambda functions in the *AWS Lambda Developer Guide*.

DynamoDB Streams and Apache Flink

You can consume Amazon DynamoDB Streams records with Apache Flink. With Amazon Managed Service for Apache Flink, you can transform and analyze streaming data in real time using Apache Flink. Apache Flink is an open-source stream processing framework for processing real-time data. The Amazon DynamoDB Streams connector for Apache Flink simplifies building and managing Apache Flink workloads and allows you to integrate applications with other AWS services.

Amazon Managed Service for Apache Flink helps you to quickly build end-to-end stream processing applications for log analytics, clickstream analytics, Internet of Things (IoT), ad tech, gaming, and more. The four most common use cases are streaming extract-transform-load (ETL), event driven applications, responsive real-time analytics, and interactive querying of data streams. For more information on writing to Apache Flink from Amazon DynamoDB Streams, see Amazon DynamoDB Streams Connector.

In-memory acceleration with DynamoDB Accelerator (DAX)

Amazon DynamoDB is designed for scale and performance. In most cases, the DynamoDB response times can be measured in single-digit milliseconds. However, there are certain use cases that require response times in microseconds. For these use cases, DynamoDB Accelerator (DAX) delivers fast response times for accessing eventually consistent data.

DAX is a DynamoDB-compatible caching service that enables you to benefit from fast in-memory performance for demanding applications. DAX addresses three core scenarios:

1. As an in-memory cache, DAX reduces the response times of eventually consistent read workloads by an order of magnitude from single-digit milliseconds to microseconds.
2. DAX reduces operational and application complexity by providing a managed service that is API-compatible with DynamoDB. Therefore, it requires only minimal functional changes to use with an existing application.
3. For read-heavy or bursty workloads, DAX provides increased throughput and potential operational cost savings by reducing the need to overprovision read capacity units. This is especially beneficial for applications that require repeated reads for individual keys.

DAX supports server-side encryption. With encryption at rest, the data persisted by DAX on disk will be encrypted. DAX writes data to disk as part of propagating changes from the primary node to read replicas. For more information, see DAX encryption at rest.

DAX also supports encryption in transit, ensuring that all requests and responses between your application and the cluster are encrypted by transport level security (TLS), and connections to the cluster can be authenticated by verification of a cluster x509 certificate. For more information, see DAX encryption in transit.

Topics

- Use cases for DAX
- DAX usage notes
- DAX: How it works
- DAX cluster components
- Creating a DAX cluster

- DAX and DynamoDB consistency models
- Developing with the DynamoDB Accelerator (DAX) client
- Managing DAX clusters
- Monitoring DynamoDB Accelerator
- DAX T3/T2 burstable instances
- DAX access control
- DAX encryption at rest
- DAX encryption in transit
- Using service-linked IAM roles for DAX
- Accessing DAX across AWS accounts
- DAX cluster sizing guide

Use cases for DAX

DAX provides access to eventually consistent data from DynamoDB tables, with microsecond latency. A Multi-AZ DAX cluster can serve millions of requests per second.

DAX is ideal for the following types of applications:

- Applications that require the fastest possible response time for reads. Some examples include real-time bidding, social gaming, and trading applications. DAX delivers fast, in-memory read performance for these use cases.
- Applications that read a small number of items more frequently than others. For example, consider an ecommerce system that has a one-day sale on a popular product. During the sale, demand for that product (and its data in DynamoDB) would sharply increase, compared to all of the other products. To mitigate the impacts of a "hot" key and a non-uniform traffic distribution, you could offload the read activity to a DAX cache until the one-day sale is over.
- Applications that are read-intensive, but are also cost-sensitive. With DynamoDB, you provision the number of reads per second that your application requires. If read activity increases, you can increase your tables' provisioned read throughput (at an additional cost). Or, you can offload the activity from your application to a DAX cluster, and reduce the number of read capacity units that you need to purchase otherwise.
- Applications that require repeated reads against a large set of data. Such an application could potentially divert database resources from other applications. For example, a long-

running analysis of regional weather data could temporarily consume all the read capacity in a DynamoDB table. This situation would negatively impact other applications that need to access the same data. With DAX, the weather analysis could be performed against cached data instead.

DAX is *not* ideal for the following types of applications:

- Applications that require strongly consistent reads (or that cannot tolerate eventually consistent reads).
- Applications that do not require microsecond response times for reads, or that do not need to offload repeated read activity from underlying tables.
- Applications that are write-intensive. High volume of writes lead to increased replication across DAX nodes in a cluster. This causes an increased consumption of resources and risk of availability issues.
- Applications without many repeated reads. DAX performs best when cache hit rates exceed 90%. Lower cache hit rates increase cache misses, which consumes more resources across the DAX cluster.

DAX usage notes

- For a list of AWS Regions where DAX is available, see Amazon DynamoDB pricing.
- DAX supports applications written in Go, Java, Node.js, Python, and .NET, using AWS-provided clients for those programming languages.
- DAX is only available for the EC2-VPC platform.
- The DAX cluster service role policy must allow the `dynamodb:DescribeTable` action in order to maintain metadata about the DynamoDB table.
- DAX clusters maintain metadata about the attribute names of items they store. That metadata is maintained indefinitely (even after the item has expired or been evicted from the cache). Applications that use an unbounded number of attribute names can, over time, cause memory exhaustion in the DAX cluster. This limitation applies only to top-level attribute names, not nested attribute names. Examples of problematic top-level attribute names include timestamps, UUIDs, and session IDs.

  This limitation applies only to attribute names, not their values. Items like the following are not a problem.

```
{
    "Id": 123,
    "Title": "Bicycle 123",
    "CreationDate": "2017-10-24T01:02:03+00:00"
}
```

But items like the following are a problem if there are enough of them and they each have a different timestamp.

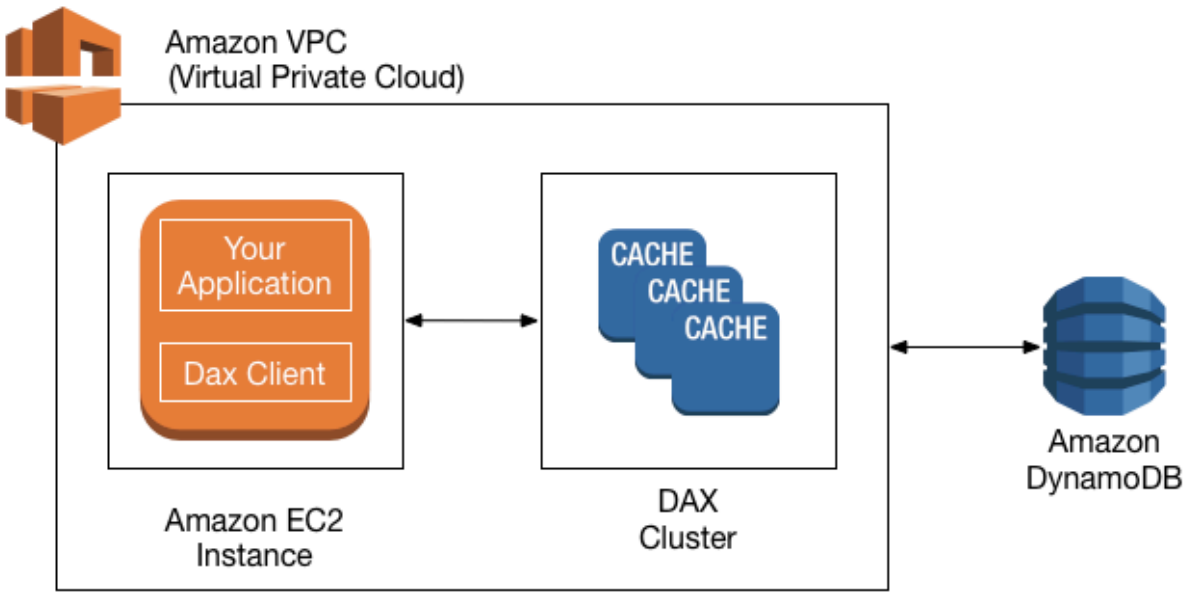
```
{
    "Id": 123,
    "Title": "Bicycle 123",
    "2017-10-24T01:02:03+00:00": "created"
}
```

DAX: How it works

Amazon DynamoDB Accelerator (DAX) is designed to run within an Amazon Virtual Private Cloud (Amazon VPC) environment. The Amazon VPC service defines a virtual network that closely resembles a traditional data center. With a VPC, you have control over its IP address range, subnets, routing tables, network gateways, and security settings. You can launch a DAX cluster in your virtual network and control access to the cluster by using Amazon VPC security groups.

> **Note**
>
> If you created your AWS account after December 4, 2013, you already have a default VPC in each AWS Region. The VPC is ready for you to use immediately—without having to perform any additional configuration steps.
> For more information, see Default VPC and default subnets in the *Amazon VPC User Guide*.

The following diagram shows a high-level overview of DAX.



To create a DAX cluster, you use the AWS Management Console. Unless you specify otherwise, your DAX cluster runs within your default VPC. To run your application, you launch an Amazon EC2 instance into your Amazon VPC. You then deploy your application (with the DAX client) on the EC2 instance.

At runtime, the DAX client directs all of your application's DynamoDB API requests to the DAX cluster. If DAX can process one of these API requests directly, it does so. Otherwise, it passes the request through to DynamoDB.

Finally, the DAX cluster returns the results to your application.

Topics

- How DAX processes requests
- Item cache
- Query cache

How DAX processes requests

A DAX cluster consists of one or more nodes. Each node runs its own instance of the DAX caching software. One of the nodes serves as the primary node for the cluster. Additional nodes (if present) serve as read replicas. For more information, see Nodes.

Your application can access DAX by specifying the endpoint for the DAX cluster. The DAX client software works with the cluster endpoint to perform intelligent load balancing and routing.

Read operations

DAX can respond to the following API calls:

- `GetItem`
- `BatchGetItem`
- `Query`
- `Scan`

If the request specifies *eventually consistent reads* (the default behavior), it tries to read the item from DAX:

- If DAX has the item available (a *cache hit*), DAX returns the item to the application without accessing DynamoDB.
- If DAX does not have the item available (a *cache miss*), DAX passes the request through to DynamoDB. When it receives the response from DynamoDB, DAX returns the results to the application. But it also writes the results to the cache on the primary node.

> **Note**
>
> If there are any read replicas in the cluster, DAX automatically keeps the replicas in sync with the primary node. For more information, see Clusters.

If the request specifies *strongly consistent reads*, DAX passes the request through to DynamoDB. The results from DynamoDB are not cached in DAX. Instead, they are simply returned to the application.

Write operations

The following DAX API operations are considered "write-through":

- `BatchWriteItem`
- `UpdateItem`
- `DeleteItem`
- `PutItem`

With these operations, data is first written to the DynamoDB table, and then to the DAX cluster. The operation is successful only if the data is successfully written to *both* the table and to DAX.

Other operations

DAX does not recognize any DynamoDB operations for managing tables (such as `CreateTable`, `UpdateTable`, and so on). If your application needs to perform these operations, it must access DynamoDB directly rather than using DAX.

For detailed information about DAX and DynamoDB consistency, see DAX and DynamoDB consistency models.

For information about how transactions work in DAX, see Using transactional APIs in DynamoDB Accelerator (DAX).

Request rate limiting

If the number of requests sent to DAX exceeds the capacity of a node, DAX limits the rate at which it accepts additional requests by returning a ThrottlingException. DAX continuously evaluates your CPU utilization to determine the volume of requests it can process while maintaining a healthy cluster state.

You can monitor the ThrottledRequestCount metric that DAX publishes to Amazon CloudWatch. If you see these exceptions regularly, you should consider scaling up your cluster.

Item cache

DAX maintains an *item cache* to store the results from `GetItem` and `BatchGetItem` operations. The items in the cache represent eventually consistent data from DynamoDB, and are stored by their primary key values.

When an application sends a `GetItem` or `BatchGetItem` request, DAX tries to read the items directly from the item cache using the specified key values. If the items are found (cache hit), DAX returns them to the application immediately. If the items are not found (cache miss), DAX sends the request to DynamoDB. DynamoDB processes the requests using eventually consistent reads and returns the items to DAX. DAX stores them in the item cache and then returns them to the application.

The item cache has a Time to Live (TTL) setting, which is 5 minutes by default. DAX assigns a timestamp to every item that it writes to the item cache. An item expires if it has remained in the cache for longer than the TTL setting. If you issue a `GetItem` request on an expired item, this is considered a cache miss, and DAX sends the `GetItem` request to DynamoDB.

> **Note**
>
> You can specify the TTL setting for the item cache when you create a new DAX cluster. For more information, see Managing DAX clusters.

DAX also maintains a least recently used (LRU) list for the item cache. The LRU list tracks when an item was first written to the cache, and when the item was last read from the cache. If the item cache becomes full, DAX evicts older items (even if they haven't expired yet) to make room for new items. The LRU algorithm is always enabled for the item cache and is not user-configurable.

If you specify zero as the *item cache* TTL setting, items in the item cache will only be refreshed due to an LRU eviction or a "write-through" operation.

For detailed information about the consistency of the item cache in DAX, see DAX item cache behavior.

Query cache

DAX also maintains a *query cache* to store the results from `Query` and `Scan` operations. The items in this cache represent result sets from queries and scans on DynamoDB tables. These result sets are stored by their parameter values.

When an application sends a `Query` or `Scan` request, DAX tries to read a matching result set from the query cache using the specified parameter values. If the result set is found (cache hit), DAX returns it to the application immediately. If the result set is not found (cache miss), DAX sends the request to DynamoDB. DynamoDB processes the requests using eventually consistent reads

and returns the result set to DAX. DAX stores it in the query cache and then returns it to the application.

> **Note**
>
> You can specify the TTL setting for the query cache when you create a new DAX cluster. For more information, see Managing DAX clusters.

DAX also maintains an LRU list for the query cache. The list tracks when a result set was first written to the cache, and when the result was last read from the cache. If the query cache becomes full, DAX evicts older result sets (even if they have not expired yet) to make room for new result sets. The LRU algorithm is always enabled for the query cache, and is not user-configurable.

If you specify zero as the *query cache* TTL setting, the query response will not be cached.

For detailed information about the consistency of the query cache in DAX, see DAX query cache behavior.

DAX cluster components

An Amazon DynamoDB Accelerator (DAX) cluster consists of AWS infrastructure components. This section describes these components and how they work together.

Topics

- Nodes
- Clusters
- Regions and availability zones
- Parameter groups
- Security groups
- Cluster ARN
- Cluster endpoint
- Node endpoints
- Subnet groups
- Events

- Maintenance window

Nodes

A *node* is the smallest building block of a DAX cluster. Each node runs an instance of the DAX software, and maintains a single replica of the cached data.

You can scale your DAX cluster in one of two ways:

- By adding more nodes to the cluster. This increases the overall read throughput of the cluster.
- By using a larger node type. Larger node types provide more capacity and can increase throughput. (You must create a new cluster with the new node type.)

Every node within a cluster is of the same node type and runs the same DAX caching software. For a list of available node types, see Amazon DynamoDB pricing.

Clusters

A *cluster* is a logical grouping of one or more nodes that DAX manages as a unit. One of the nodes in the cluster is designated as the *primary* node, and the other nodes (if any) are *read replicas*.

The primary node is responsible for the following:

- Fulfilling application requests for cached data.
- Handling write operations to DynamoDB.
- Evicting data from the cache according to the cluster's eviction policy.

When changes are made to cached data on the primary node, DAX propagates the changes to all of the read replica nodes using replication logs. After the confirmation is received from all read replicas, DynamoDB deletes the replication logs from the primary node.

A DAX cluster can support up to 11 nodes per cluster (the primary node plus a maximum of 10 read replicas).

Read replicas are responsible for the following:

- Fulfilling application requests for cached data.
- Evicting data from the cache according to the cluster's eviction policy.

However, unlike the primary node, read replicas don't write to DynamoDB.

Read replicas serve two additional purposes:

- **Scalability**. If you have a large number of application clients that need to access DAX concurrently, you can add more replicas for read-scaling. DAX spreads the load evenly across all the nodes in the cluster. (Another way to increase throughput is to use larger cache node types.)
- **High availability**. In the event of a primary node failure, DAX automatically fails over to a read replica and designates it as the new primary. If a replica node fails, other nodes in the DAX cluster can still serve requests until the failed node can be recovered. For maximum fault tolerance, you should deploy read replicas in separate Availability Zones. This configuration ensures that your DAX cluster can continue to function, even if an entire Availability Zone becomes unavailable.

> **Important**
>
> For production usage, we strongly recommend using DAX with at least three nodes, where each node is placed in different Availability Zones. Three nodes are required for a DAX cluster to be fault-tolerant.
> A DAX cluster can be deployed with one or two nodes for development or test workloads. One-node and two-node clusters are not fault-tolerant, and we don't recommend using fewer than three nodes for production use. If a one-node or two-node cluster encounters software or hardware errors, the cluster can become unavailable or lose cached data.

> **Important**
>
> A DAX cluster supports a maximum of 500 DynamoDB tables. If you go beyond 500 tables, your cluster may experience degradation in availability and performance.

Regions and availability zones

A DAX cluster in an AWS Region can only interact with DynamoDB tables that are in the same Region. For this reason, ensure that you launch your DAX cluster in the correct Region. If you have DynamoDB tables in other Regions, you must launch DAX clusters in those Regions too.

Each Region is designed to be completely isolated from the other Regions. Within each Region are multiple Availability Zones. By launching your nodes in different Availability Zones, you can achieve the greatest possible fault tolerance.

> **Important**
>
> Don't place all of your cluster's nodes in a single Availability Zone. In this configuration, your DAX cluster becomes unavailable if there is an Availability Zone failure.
> For production usage, we strongly recommend using DAX with at least three nodes, where each node is placed in different Availability Zones. Three nodes are required for a DAX cluster to be fault-tolerant.
> A DAX cluster can be deployed with one or two nodes for development or test workloads. One- and two-node clusters are not fault-tolerant, and we don't recommend using fewer than three nodes for production use. If a one- or two-node cluster encounters software or hardware errors, the cluster can become unavailable or lose cached data.

Parameter groups

Parameter groups are used to manage runtime settings for DAX clusters. DAX has several parameters that you can use to optimize performance (such as defining a TTL policy for cached data). A parameter group is a named set of parameters that you can apply to a cluster. You can thereby ensure that all the nodes in that cluster are configured in exactly the same way.

Security groups

A DAX cluster runs in an Amazon Virtual Private Cloud (Amazon VPC) environment. This environment is a virtual network that is dedicated to your AWS account and is isolated from other VPCs. A *security group* acts as a virtual firewall for your VPC, allowing you to control inbound and outbound network traffic.

When you launch a cluster in your VPC, you add an *ingress* rule to your security group to allow incoming network traffic. The ingress rule specifies the protocol (TCP) and port number (8111) for your cluster. After you add this rule to your security group, the applications that are running within your VPC can access the DAX cluster.

Cluster ARN

Every DAX cluster is assigned an *Amazon Resource Name* (ARN). The ARN format is as follows.

```
arn:aws:dax:region:accountID:cache/clusterName
```

You use the cluster ARN in an IAM policy to define permissions for DAX API operations. For more information, see DAX access control.

Cluster endpoint

Every DAX cluster provides a *cluster endpoint* for use by your application. By accessing the cluster using its endpoint, your application does not need to know the hostnames and port numbers of individual nodes in the cluster. Your application automatically "knows" all the nodes in the cluster, even if you add or remove read replicas.

The following is an example of a cluster endpoint in the us-east-1 region that is not configured to use encryption in transit.

```
dax://my-cluster.l6fzcv.dax-clusters.us-east-1.amazonaws.com
```

The following is an example of a cluster endpoint in the same region that is configured to use encryption in transit.

```
daxs://my-encrypted-cluster.l6fzcv.dax-clusters.us-east-1.amazonaws.com
```

Node endpoints

Each of the individual nodes in a DAX cluster has its own hostname and port number. The following is an example of a *node endpoint*.

```
myDAXcluster-a.2cmrwl.clustercfg.dax.use1.cache.amazonaws.com:8111
```

Your application can access a node directly by using its endpoint. However, we recommend that you treat the DAX cluster as a single unit and access it using the cluster endpoint instead. The cluster endpoint insulates your application from having to maintain a list of nodes and keep that list up to date when you add or remove nodes from the cluster.

Subnet groups

Access to DAX cluster nodes is restricted to applications running on Amazon EC2 instances within an Amazon VPC environment. You can use *subnet groups* to grant cluster access from Amazon EC2 instances running on specific subnets. A subnet group is a collection of subnets (typically private) that you can designate for your clusters running in an Amazon VPC environment.

When you create a DAX cluster, you must specify a subnet group. DAX uses that subnet group to select a subnet and IP addresses within that subnet to associate with your nodes.

Events

DAX records significant events within your clusters, such as a failure to add a node, success in adding a node, or changes to security groups. By monitoring key events, you can know the current state of your clusters and, depending upon the event, be able to take corrective action. You can access these events using the AWS Management Console or the `DescribeEvents` action in the DAX management API.

You can also request that notifications be sent to a specific Amazon Simple Notification Service (Amazon SNS) topic. Then you will know immediately when an event occurs in your DAX cluster.

Maintenance window

Every cluster has a weekly maintenance window to apply system changes. As changes are applied sequentially, an existing node is replaced and a new node with the applied changes is added to the cluster. During this period, your application might observe transient errors or throttles. Therefore, we recommend that you schedule the maintenance window during your lowest usage time and adjust this schedule periodically as needed. You can specify a time range of up to 24 hours in duration during which any maintenance activities that you request should occur.

If you don't specify a preferred maintenance window when you create or modify a cache cluster, DAX assigns a 60-minute maintenance window on a random weekday. This 60-minute maintenance window is randomly selected from an 8-hour block of time for each AWS Region. The following table lists the time blocks for each Region from which the default maintenance windows are assigned.

Region code	Region name	Maintenance window
ap-northeast-1	Asia Pacific (Tokyo) Region	13:00–21:00 UTC
ap-southeast-1	Asia Pacific (Singapore) Region	14:00–22:00 UTC
ap-southeast-2	Asia Pacific (Sydney) Region	12:00–20:00 UTC
ap-south-1	Asia Pacific (Mumbai) Region	17:30–1:30 UTC

Region code	Region name	Maintenance window
cn-northwest-1	China (Ningxia) Region	23:00–07:00 UTC
cn-north-1	China (Beijing) Region	14:00–22:00 UTC
eu-central-1	Europe (Frankfurt) Region	23:00–07:00 UTC
eu-north-1	Europe (Stockholm) Region	01:00–09:00 UTC
eu-south-2	Europe (Spain) Region	21:00–05:00 UTC
eu-west-1	Europe (Ireland) Region	22:00–06:00 UTC
eu-west-2	Europe (London) Region	23:00–07:00 UTC
eu-west-3	Europe (Paris) Region	23:00–07:00 UTC
sa-east-1	South America (São Paulo) Region	01:00–09:00 UTC
us-east-1	US East (N. Virginia) Region	03:00–11:00 UTC
us-east-2	US East (Ohio) Region	23:00–07:00 UTC
us-west-1	US West (N. California) Region	06:00–14:00 UTC
us-west-2	US West (Oregon) Region	06:00–14:00 UTC

Creating a DAX cluster

This section walks you through the first-time setup and usage of Amazon DynamoDB Accelerator (DAX) in your default Amazon Virtual Private Cloud (Amazon VPC) environment. You can create your first DAX cluster using either the AWS Command Line Interface (AWS CLI) or the AWS Management Console.

After you create your DAX cluster, you can access it from an Amazon EC2 instance running in the same VPC. You can then use your DAX cluster with an application program. For more information, see Developing with the DynamoDB Accelerator (DAX) client.

Topics

- Creating an IAM service role for DAX to access DynamoDB
- DAX and IPv6
- Creating a DAX cluster using the AWS CLI
- Creating a DAX cluster using the AWS Management Console

Creating an IAM service role for DAX to access DynamoDB

For your DAX cluster to access DynamoDB tables on your behalf, you must create a *service role*. A service role is an AWS Identity and Access Management (IAM) role that authorizes an AWS service to act on your behalf. The service role allows DAX to access your DynamoDB tables, as if you were accessing those tables yourself. You must create the service role before you can create the DAX cluster.

If you are using the console, the workflow for creating a cluster checks for the presence of a pre-existing DAX service role. If none is found, the console creates a new service role for you. For more information, see the section called “Step 2: Create a DAX cluster”.

If you are using the AWS CLI, you must specify a DAX service role that you have created previously. Otherwise, you need to create a new service role beforehand. For more information, see Step 1: Create an IAM service role for DAX to access DynamoDB using the AWS CLI.

Permissions required to create a service role

The AWS managed `AdministratorAccess` policy provides all the permissions needed for creating a DAX cluster and a service role. If your user has `AdministratorAccess` attached, no further action is needed.

Otherwise, you must add the following permissions to your IAM policy so that your user can create the service role:

- `iam:CreateRole`
- `iam:CreatePolicy`
- `iam:AttachRolePolicy`
- `iam:PassRole`

Attach these permissions to the user who is trying to perform the action.

> **Note**
>
> The `iam:CreateRole`, `iam:CreatePolicy`, `iam:AttachRolePolicy`, and `iam:PassRole` permissions are not included in the AWS managed policies for DynamoDB. This is by design because these permissions provide the possibility of privilege escalation: That is, a user could use these permissions to create a new administrator policy and then attach that policy to an existing role. For this reason, you (the administrator of your DAX cluster) must explicitly add these permissions to your policy.

Troubleshooting

If your user policy is missing the `iam:CreateRole`, `iam:CreatePolicy`, and `iam:AttachPolicy` permissions, you will get error messages. The following table lists these messages and describes how to correct the problems.

If you see this error message...	Do the following:
`User: arn:aws:iam:: accountID :user/userName is not authorized to perform: iam:CreateRole on resource: arn:aws:iam:: accountID :role/service-role/ roleName`	Add `iam:CreateRole` to your user policy.
`User: arn:aws:iam:: accountID :user/userName is not authorized to perform: iam:CreatePolicy on resource: policy policyName`	Add `iam:CreatePolicy` to your user policy.
`User: arn:aws:iam:: accountID :user/userName is not authorized to perform: iam:AttachRolePolicy on resource: role daxServiceRole`	Add `iam:AttachRolePolicy` to your user policy.

For more information about the IAM policies required for DAX cluster administration, see DAX access control.

DAX and IPv6

DynamoDB DAX now supports IPv6 addressing, that allows you to create clusters that operate in IPv4-only, IPv6-only, or dual-stack networking modes. This helps in enhancing networking capabilities to meet evolving infrastructure requirements.

Network Types:

You can create clusters with the following network types:

- IPv4-only
- IPv6-only
- Dual-stack (supports both IPv4 and IPv6)

Key Features:

With IPv6 support, you can do the following:

- **Network configuration options:**
  - IPv4-only and dual-stack clusters on `dual_stack subnets`.
  - IPv6-only clusters on IPv6-only subnets.
- **Subnet group management:**
  - Create subnet groups with IPv4-only, IPv6-only, or dual-stack support
  - Modify existing subnet groups with additional VPC subnets
  - Add IPv6-only subnets to IPv6-configured subnet groups
  - Add IPv4 or dual-stack subnets to IPv4 and dual-stack configured groups
- **Client configuration:**
  - When making data plane calls, you can set preferred IP protocol for dual_stack clusters using:
    - `ip_discovery` parameter in Python SDK
    - `ipDiscovery` parameter in other SDKs
  - Default: IPv4 when protocol preference not specified

Before implementing IPv6 in your DAX clusters, you must consider the following:

- Network type cannot be changed after cluster creation

- For dual-stack clusters, the `ip_discovery/ipDiscovery` parameter in the client configuration determines which IP protocol to use (IPv4 or IPv6)
- Different applications can connect to the same dual-stack cluster using different IP protocols based on their configuration

Example client configuration

```
DynamoDbAsyncClient client = ClusterDaxAsyncClient.builder()
        .overrideConfiguration(Configuration.builder()
            .url(endpoint)             // DAX cluster endpoint
            .ipDiscovery(ipDiscovery)       // IP discovery type (IPv4 or IPv6)
            .build())
        .build();
```

Important

When you use resource-based IAM policies to restrict IP addresses for DynamoDB tables in IPv6-only environments with DAX, you must create an exception for your DAX cluster's IAM role if you block the IPv4 address space (`0.0.0.0/0`). Add an `ArnNotEquals` condition to your policy that specifically allows access for the DAX cluster's IAM role while maintaining IP-based restrictions for other access paths. Without this exception, DAX cannot access your DynamoDB table.
For example:

Example

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Deny",
      "Principal": "*",
      "Action": "dynamodb:PutItem",
      "Resource": "arn:aws:dynamodb:us-
east-1:123456789012:table/MusicCollection",
      "Condition": {
        "ArnNotEquals": {
```

```
          "aws:PrincipalArn": "arn:aws:iam::123456789012:role/
DAXServiceRoleForDynamoDBAccess"
        },
        "IpAddress": {
          "aws:SourceIp": "0.0.0.0/0"
        }
      }
    }
  ]
}
```

Creating a DAX cluster using the AWS CLI

This section describes how to create an Amazon DynamoDB Accelerator (DAX) cluster using the AWS Command Line Interface (AWS CLI). If you haven't already done so, you must install and configure the AWS CLI. To do this, see the following instructions in the *AWS Command Line Interface User Guide*:

- Installing the AWS CLI
- Configuring the AWS CLI

> **Important**
>
> To manage DAX clusters using the AWS CLI, install or upgrade to version 1.11.110 or higher.

All of the AWS CLI examples use the `us-west-2` Region and fictitious account IDs.

Topics

- Step 1: Create an IAM service role for DAX to access DynamoDB using the AWS CLI
- Step 2: Create a subnet group
- Step 3: Create a DAX cluster using the AWS CLI
- Step 4: Configure security group inbound rules using the AWS CLI

Step 1: Create an IAM service role for DAX to access DynamoDB using the AWS CLI

Before you can create an Amazon DynamoDB Accelerator (DAX) cluster, you must create a service role for it. A *service role* is an AWS Identity and Access Management (IAM) role that authorizes an AWS service to act on your behalf. The service role allows DAX to access your DynamoDB tables as if you were accessing those tables yourself.

In this step, you create an IAM policy and then attach that policy to an IAM role. This enables you to assign the role to a DAX cluster so that it can perform DynamoDB operations on your behalf.

To create an IAM service role for DAX

1. Create a file named `service-trust-relationship.json` with the following contents.

   JSON

   ```
   {
       "Version":"2012-10-17",
       "Statement": [
          {
               "Effect": "Allow",
               "Principal": {
                   "Service": "dax.amazonaws.com"
               },
               "Action": "sts:AssumeRole"
           }
       ]
   }
   ```

2. Create the service role.

   ```
   aws iam create-role \
       --role-name DAXServiceRoleForDynamoDBAccess \
       --assume-role-policy-document file://service-trust-relationship.json
   ```

3. Create a file named `service-role-policy.json` with the following contents.

   JSON

   ```
   {
       "Version":"2012-10-17",
   ```

```
    "Statement": [
        {
            "Action": [
                "dynamodb:DescribeTable",
                "dynamodb:PutItem",
                "dynamodb:GetItem",
                "dynamodb:UpdateItem",
                "dynamodb:DeleteItem",
                "dynamodb:Query",
                "dynamodb:Scan",
                "dynamodb:BatchGetItem",
                "dynamodb:BatchWriteItem",
                "dynamodb:ConditionCheckItem"
            ],
            "Effect": "Allow",
            "Resource": [
                "arn:aws:dynamodb:us-west-2:111122223333:*"
            ]
        }
    ]
}
```

Replace *accountID* with your AWS account ID. To find your AWS account ID, in the upper-right corner of the console, choose your login ID. Your AWS account ID appears in the drop-down menu.

In the Amazon Resource Name (ARN) in the example, *accountID* must be a 12-digit number. Don't use hyphens or any other punctuation.

4. Create an IAM policy for the service role.

```
aws iam create-policy \
    --policy-name DAXServicePolicyForDynamoDBAccess \
    --policy-document file://service-role-policy.json
```

In the output, note the ARN for the policy that you created, as in the following example.

```
arn:aws:iam::123456789012:policy/DAXServicePolicyForDynamoDBAccess
```

5. Attach the policy to the service role. Replace *arn* in the following code with the actual role ARN from the previous step.

```
aws iam attach-role-policy \
    --role-name DAXServiceRoleForDynamoDBAccess \
    --policy-arn arn
```

Next, you specify a subnet group for your default VPC. A *subnet group* is a collection of one or more subnets within your VPC. See Step 2: Create a subnet group.

Step 2: Create a subnet group

Follow this procedure to create a subnet group for your Amazon DynamoDB Accelerator (DAX) cluster using the AWS Command Line Interface (AWS CLI).

> **Note**
>
> If you already created a subnet group for your default VPC, you can skip this step.

DAX is designed to run within an Amazon Virtual Private Cloud environment (Amazon VPC). If you created your AWS account after December 4, 2013, you already have a default VPC in each AWS Region. For more information, see Default VPC and default subnets in the *Amazon VPC User Guide*.

To create a subnet group

1. To determine the identifier for your default VPC, enter the following command.

   ```
   aws ec2 describe-vpcs
   ```

   In the output, note the identifier for your default VPC, as in the following example.

   `vpc-12345678`

2. Determine the subnet IDs associated with your default VPC. Replace *vpcID* with your actual VPC ID—for example, `vpc-12345678`.

   ```
   aws ec2 describe-subnets \
       --filters "Name=vpc-id,Values=vpcID" \
       --query "Subnets[*].SubnetId"
   ```

   In the output, note the subnet identifiers—for example, `subnet-11111111`.

3. Create the subnet group. Ensure that you specify at least one subnet ID in the `--subnet-ids` parameter.

```
aws dax create-subnet-group \
    --subnet-group-name my-subnet-group \
    --subnet-ids subnet-11111111 subnet-22222222 subnet-33333333 subnet-44444444
```

To create the cluster, see Step 3: Create a DAX cluster using the AWS CLI.

Step 3: Create a DAX cluster using the AWS CLI

Follow this procedure to use the AWS Command Line Interface (AWS CLI) to create an Amazon DynamoDB Accelerator (DAX) cluster in your default Amazon VPC.

To create a DAX cluster

1. Get the Amazon Resource Name (ARN) for your service role.

```
aws iam get-role \
    --role-name DAXServiceRoleForDynamoDBAccess \
    --query "Role.Arn" --output text
```

In the output, note the service role ARN, as in the following example.

```
arn:aws:iam::123456789012:role/DAXServiceRoleForDynamoDBAccess
```

2. Create the DAX cluster. Replace *roleARN* with the ARN from the previous step.

```
aws dax create-cluster \
    --cluster-name mydaxcluster \
    --node-type dax.r4.large \
    --replication-factor 3 \
    --iam-role-arn roleARN \
    --subnet-group my-subnet-group \
    --sse-specification Enabled=true \
    --region us-west-2
```

All of the nodes in the cluster are of type `dax.r4.large` (`--node-type`). There are three nodes (`--replication-factor`)—one primary node and two replicas.

> **Note**
>
> Since `sudo` and `grep` are reserved keywords, you cannot create a DAX cluster with these words in the cluster name. For example, `sudo` and `sudocluster` are invalid cluster names.

To view the cluster status, enter the following command.

```
aws dax describe-clusters
```

The status is shown in the output—for example, `"Status": "creating"`.

> **Note**
>
> Creating the cluster takes several minutes. When the cluster is ready, its status changes to `available`. In the meantime, proceed to Step 4: Configure security group inbound rules using the AWS CLI and follow the instructions there.

Step 4: Configure security group inbound rules using the AWS CLI

The nodes in your Amazon DynamoDB Accelerator (DAX) cluster use the default security group for your Amazon VPC. For the default security group, you must authorize inbound traffic on TCP port 8111 for unencrypted clusters or port 9111 for encrypted clusters. This allows Amazon EC2 instances in your Amazon VPC to access your DAX cluster.

> **Note**
>
> If you launched your DAX cluster with a different security group (other than `default`), you must perform this procedure for that group instead.

To configure security group inbound rules

1. To determine the default security group identifier, enter the following command. Replace *`vpcID`* with your actual VPC ID (from Step 2: Create a subnet group).

```
aws ec2 describe-security-groups \
    --filters Name=vpc-id,Values=vpcID Name=group-name,Values=default \
    --query "SecurityGroups[*].{GroupName:GroupName,GroupId:GroupId}"
```

In the output, note the security group identifier—for example, `sg-01234567`.

2. Then enter the following. Replace *sgID* with your actual security group identifier. Use port 8111 for unencrypted clusters and 9111 for encrypted clusters.

```
aws ec2 authorize-security-group-ingress \
    --group-id sgID --protocol tcp --port 8111
```

Creating a DAX cluster using the AWS Management Console

This section describes how to create an Amazon DynamoDB Accelerator (DAX) cluster using the AWS Management Console.

Topics

- Step 1: Create a subnet group using the AWS Management Console
- Step 2: Create a DAX cluster using the AWS Management Console
- Step 3: Configure security group inbound rules using the AWS Management Console

Step 1: Create a subnet group using the AWS Management Console

Follow this procedure to create a subnet group for your Amazon DynamoDB Accelerator (DAX) cluster using the AWS Management Console.

> **Note**
>
> If you already created a subnet group for your default VPC, you can skip this step.

DAX is designed to run within an Amazon Virtual Private Cloud (Amazon VPC) environment. If you created your AWS account after December 4, 2013, you already have a default VPC in each AWS Region. For more information, see Default VPC and default subnets in the *Amazon VPC User Guide*.

As part of the creation process for a DAX cluster, you must specify a *subnet group*. A subnet group is a collection of one or more subnets within your VPC. When you create your DAX cluster, the nodes are deployed to the subnets within the subnet group.

> **Note**
>
> The VPC having this DAX cluster can contain other resources and even VPC endpoints for the other services except VPC endpoint for ElastiCache and can result in error for the DAX cluster operations.

To create a subnet group

1. Open the DynamoDB console at https://console.aws.amazon.com/dynamodb/.
2. In the navigation pane, under **DAX**, choose **Subnet groups**.
3. Choose **Create subnet group**.
4. In the **Create subnet group** window, do the following:
   a. **Name**—Enter a short name for the subnet group.
   b. **Description**—Enter a description for the subnet group.
   c. **VPC ID**—Choose the identifier for your Amazon VPC environment.
   d. **Subnets**—Choose one or more subnets from the list.

   > **Note**
   >
   > The subnets are distributed among multiple Availability Zones. If you plan to create a multi-node DAX cluster (a primary node and one or more read replicas), we recommend that you choose multiple subnet IDs. Then DAX can deploy the cluster nodes into multiple Availability Zones. If an Availability Zone becomes unavailable, DAX automatically fails over to a surviving Availability Zone. Your DAX cluster will continue to function without interruption.

   When the settings are as you want them, choose **Create subnet group**.

To create the cluster, see Step 2: Create a DAX cluster using the AWS Management Console.

Step 2: Create a DAX cluster using the AWS Management Console

Follow this procedure to create an Amazon DynamoDB Accelerator (DAX) cluster in your default Amazon VPC.

To create a DAX cluster

1. Open the DynamoDB console at https://console.aws.amazon.com/dynamodb/.
2. In the navigation pane, under **DAX**, choose **Clusters**.
3. Choose **Create cluster**.
4. In the **Create cluster** window, do the following:

   a. **Cluster name**—Enter a short name for your DAX cluster.

   > **Note**
   >
   > Since `sudo` and `grep` are reserved keywords, you cannot create a DAX cluster with these words in the cluster name. For example, `sudo` and `sudocluster` are invalid cluster names.

   b. **Cluster description**—Enter a description for the cluster.

   c. **Node types**—Choose the node type for all of the nodes in the cluster.

   d. **Cluster size**—Choose the number of nodes in the cluster. A cluster consists of one primary node and up to nine read replicas.

   > **Note**
   >
   > If you want to create a single-node cluster, choose **1**. Your cluster will consist of one primary node.
   > If you want to create a multi-node cluster, choose a number between **3** (one primary and two read replicas) and **10** (one primary and nine read replicas).
   >
   > > **Important**
   > >
   > > For production usage, we strongly recommend using DAX with at least three nodes, where each node is placed in a different Availability Zone. Three nodes are required for a DAX cluster to be fault-tolerant.

A DAX cluster can be deployed with one or two nodes for development or test workloads. One- and two-node clusters are not fault-tolerant, and we don't recommend using fewer than three nodes for production use. If a one- or two-node cluster encounters software or hardware errors, the cluster can become unavailable or lose cached data.

e. Choose **Next**.

f. **Subnet group**—Select **Choose existing** and choose the subnet group that you created in Step 1: Create a subnet group using the AWS Management Console.

g. **Access control**—Choose the **default** security group.

h. **Availability Zones (AZ)**—Choose **Automatic**.

i. Choose next.

j. **IAM service role for DynamoDB access**—Choose **Create new**, and enter the following information:

- **IAM role name**—Enter a name for an IAM role, for example, `DAXServiceRole`. The console creates a new IAM role, and your DAX cluster assumes this role at runtime.
- Select the box next to **Create policy**.
- **IAM role policy**—Choose **Read/Write**. This allows the DAX cluster to perform read and write operations in DynamoDB.
- **New IAM policy name**—This field will populate as you enter the IAM role name. You can also enter a name for an IAM policy, for example, `DAXServicePolicy`. The console creates a new IAM policy and attaches the policy to the IAM role.
- **Access to DynamoDB tables**—Choose **All tables**.

k. **Encryption**—Choose **Turn on encryption at rest** and **Turn on encryption in transit** For more information, see DAX encryption at rest and DAX encryption in transit.

A separate service role for DAX to access Amazon EC2 is also required. DAX automatically creates this service role for you. For more information, see Using service-linked roles for DAX.

5. When the settings are as you want them, choose **Next**.

6. **Parameter group**—Choose **Choose existing**.

7. **Maintenance window**—Choose **No preference** if you don't have a preference when software upgrades are applied, or choose **Specify time window** and provide the **Weekday**, **Time(UTC)** and **Start within (hours)** options to schedule your maintenance window.
8. **Tags**—Choose **Add new tag** to enter a key/value pair for tagging purposes.
9. Choose **Next**.

On the **Review and create** screen, you can review all of the settings. If you are ready to create the cluster, choose **Create cluster**.

On the **Clusters** screen, your DAX cluster will be listed with a status of **Creating**.

> **Note**
>
> Creating the cluster takes several minutes. When the cluster is ready, its status changes to **Available**.
> In the meantime, proceed to Step 3: Configure security group inbound rules using the AWS Management Console and follow the instructions there.

Step 3: Configure security group inbound rules using the AWS Management Console

Your Amazon DynamoDB Accelerator (DAX) cluster communicates via TCP port 8111 (for unencrypted clusters) or 9111 (for encrypted clusters), so you must authorize inbound traffic on that port. This allows Amazon EC2 instances in your Amazon VPC to access your DAX cluster.

> **Note**
>
> If you launched your DAX cluster with a different security group (other than `default`), you must perform this procedure for that group instead.

To configure security group inbound rules

1. Open the Amazon EC2 console at https://console.aws.amazon.com/ec2/.
2. In the navigation pane, choose **Security Groups**.
3. Choose the **default** security group. On the **Actions** menu, choose **Edit inbound rules**.

4. Choose **Add Rule**, and enter the following information:
   - **Port Range**—Enter **8111** (if your cluster is unencrypted) or **9111** (if your cluster is encrypted).
   - **Source**—Leave this as **Custom**, and choose the search field to the right. A drop-down menu will be displayed. Choose the identifier for your default security group.
5. Choose **Save rules** to save your changes.
6. To update the name in the console, go to the **Name** property and choose the **Edit** option that is displayed.

DAX and DynamoDB consistency models

Amazon DynamoDB Accelerator (DAX) is a write-through caching service that is designed to simplify the process of adding a cache to DynamoDB tables. Because DAX operates separately from DynamoDB, it is important that you understand the consistency models of both DAX and DynamoDB to ensure that your applications behave as you expect.

In many use cases, the way that your application uses DAX affects the consistency of data within the DAX cluster, and the consistency of data between DAX and DynamoDB.

Topics

- Consistency among DAX cluster nodes
- DAX item cache behavior
- DAX query cache behavior
- Strongly consistent and transactional reads
- Negative caching
- Strategies for writes

Consistency among DAX cluster nodes

To achieve high availability for your application, we recommend that you provision your DAX cluster with at least three nodes. Then place those nodes in multiple Availability Zones within a Region.

When your DAX cluster is running, it replicates the data among all of the nodes in the cluster (assuming that you provisioned more than one node). Consider an application that performs a

successful `UpdateItem` using DAX. This action causes the item cache in the primary node to be modified with the new value. That value is then replicated to all the other nodes in the cluster. This replication is eventually consistent and usually takes less than one second to complete.

In this scenario, it's possible for two clients to read the same key from the same DAX cluster but receive different values, depending on the node that each client accessed. The nodes are all consistent when the update has been fully replicated throughout all the nodes in the cluster. (This behavior is similar to the eventually consistent nature of DynamoDB.)

If you are building an application that uses DAX, that application should be designed so that it can tolerate eventually consistent data.

DAX item cache behavior

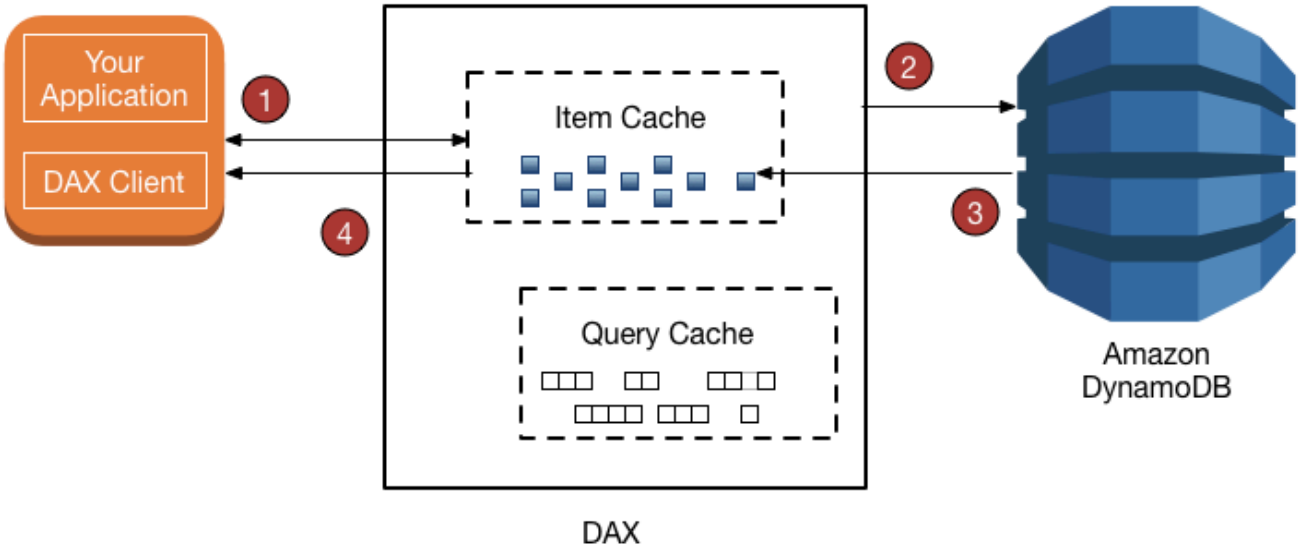
Every DAX cluster has two distinct caches—an *item* cache and a *query* cache. For more information, see DAX: How it works.

This section addresses the consistency implications of reading from and writing to the DAX item cache.

Consistency of reads

With DynamoDB, the `GetItem` operation performs an eventually consistent read by default. Suppose that you use `UpdateItem` with the DynamoDB client. If you then try to read the same item immediately afterward, you might see the data as it appeared before the update. This is due to propagation delay across all the DynamoDB storage locations. Consistency is usually reached within seconds. So if you retry the read, you will likely see the updated item.

When you use `GetItem` with the DAX client, the operation (in this case, an eventually consistent read) proceeds as shown following.



1. The DAX client issues a `GetItem` request. DAX tries to read the requested item from the item cache. If the item is in the cache (*cache hit*), DAX returns it to the application.
2. If the item is not available (*cache miss*), DAX performs an eventually consistent `GetItem` operation against DynamoDB.
3. DynamoDB returns the requested item, and DAX stores it in the item cache.
4. DAX returns the item to the application.
5. (Not shown) If the DAX cluster contains more than one node, the item is replicated to all the other nodes in the cluster.

The item remains in the DAX item cache, subject to the Time to Live (TTL) setting and the least recently used (LRU) algorithm for the cache. For more information, see DAX: How it works.

However, during this period, DAX doesn't re-read the item from DynamoDB. If someone else updates the item using a DynamoDB client, bypassing DAX entirely, a `GetItem` request using the DAX client yields different results from the same `GetItem` request using the DynamoDB client. In this scenario, DAX and DynamoDB hold inconsistent values for the same key until the TTL for the DAX item expires.

If an application modifies data in an underlying DynamoDB table, bypassing DAX, the application needs to anticipate and tolerate data inconsistencies that might arise.

> **Note**
>
> In addition to `GetItem`, the DAX client also supports `BatchGetItem` requests. `BatchGetItem` is essentially a wrapper around one or more `GetItem` requests, so DAX treats each of these as an individual `GetItem` operation.

Consistency of writes

DAX is a write-through cache, which simplifies the process of keeping the DAX item cache consistent with the underlying DynamoDB tables.

The DAX client supports the same write API operations as DynamoDB (`PutItem`, `UpdateItem`, `DeleteItem`, `BatchWriteItem`, and `TransactWriteItems`). When you use these operations with the DAX client, the items are modified in both DAX and DynamoDB. DAX updates the items in its item cache, regardless of the TTL value for these items.

For example, suppose that you issue a `GetItem` request from the DAX client to read an item from the `ProductCatalog` table. (The partition key is `Id`, and there is no sort key.) You retrieve the item whose `Id` is `101`. The `QuantityOnHand` value for that item is `42`. DAX stores the item in its item cache with a specific TTL. For this example, assume that the TTL is 10 minutes. Then, 3 minutes later, another application uses the DAX client to update the same item so that its `QuantityOnHand` value is now `41`. Assuming that the item is not updated again, any subsequent reads of the same item during the next 10 minutes return the cached value for `QuantityOnHand` (`41`).

How DAX processes writes

DAX is intended for applications that require high-performance reads. As a write-through cache, DAX passes your writes through to DynamoDB synchronously, then automatically and asynchronously replicates resulting updates to your item cache across all nodes in the cluster. You don't need to manage cache invalidation logic because DAX handles it for you.

DAX supports the following write operations: `PutItem`, `UpdateItem`, `DeleteItem`, `BatchWriteItem`, and `TransactWriteItems`.

When you send a `PutItem`, `UpdateItem`, `DeleteItem`, or `BatchWriteItem` request to DAX, the following occurs:

- DAX sends the request to DynamoDB.
- DynamoDB replies to DAX, confirming that the write succeeded.
- DAX writes the item to its item cache.
- DAX returns success to the requester.

When you send a `TransactWriteItems` request to DAX, the following occurs:

- DAX sends the request to DynamoDB.
- DynamoDB replies to DAX, confirming that the transaction completed.
- DAX returns success to the requester.
- In the background, DAX makes a `TransactGetItems` request for each item in the `TransactWriteItems` request to store the item in the item cache. `TransactGetItems` is used to ensure serializable isolation.

If a write to DynamoDB fails for any reason, including throttling, the item is not cached in DAX. The exception for the failure is returned to the requester. This ensures that data is not written to the DAX cache unless it is first written successfully to DynamoDB.

> **Note**
>
> Every write to DAX alters the state of the item cache. However, writes to the item cache don't affect the query cache. (The DAX item cache and query cache serve different purposes, and operate independently from one another.)

DAX query cache behavior

DAX caches the results from `Query` and `Scan` requests in its query cache. However, these results don't affect the item cache at all. When your application issues a `Query` or `Scan` request with DAX, the result set is saved in the query cache—not in the item cache. You can't "warm up" the item cache by performing a `Scan` operation because the item cache and query cache are separate entities.

Consistency of query-update-query

Updates to the item cache, or to the underlying DynamoDB table, do not invalidate or modify the results stored in the query cache.

To illustrate, consider the following scenario. An application is working with the `DocumentRevisions` table, which has `DocId` as its partition key and `RevisionNumber` as its sort key.

1. A client issues a `Query` for `DocId` `101`, for all items with `RevisionNumber` greater than or equal to 5. DAX stores the result set in the query cache and returns the result set to the user.
2. The client issues a `PutItem` request for `DocId` `101` with a `RevisionNumber` value of `20`.
3. The client issues the same `Query` as described in step 1 (`DocId` `101` and `RevisionNumber` >= 5).

In this scenario, the cached result set for the `Query` issued in step 3 is identical to the result set that was cached in step 1. The reason is that DAX does not invalidate `Query` or `Scan` result sets based on updates to individual items. The `PutItem` operation from step 2 is only reflected in the DAX query cache when the TTL for the `Query` expires.

Your application should consider the TTL value for the query cache and how long your application can tolerate inconsistent results between the query cache and the item cache.

Strongly consistent and transactional reads

To perform a strongly consistent `GetItem`, `BatchGetItem`, `Query`, or `Scan` request, you set the `ConsistentRead` parameter to true. DAX passes strongly consistent read requests to DynamoDB. When it receives a response from DynamoDB, DAX returns the results to the client, but it does not cache the results. DAX can't serve strongly consistent reads by itself because it's not tightly coupled to DynamoDB. For this reason, any subsequent reads from DAX would have to be eventually consistent reads. And any subsequent strongly consistent reads would have to be passed through to DynamoDB.

DAX handles `TransactGetItems` requests the same way it handles strongly consistent reads. DAX passes all `TransactGetItems` requests to DynamoDB. When it receives a response from DynamoDB, DAX returns the results to the client, but it doesn't cache the results.

Negative caching

DAX supports negative cache entries in both the item cache and the query cache. A *negative cache entry* occurs when DAX can't find requested items in an underlying DynamoDB table. Instead of generating an error, DAX caches an empty result and returns that result to the user.

For example, suppose that an application sends a `GetItem` request to a DAX cluster, and that there is no matching item in the DAX item cache. This causes DAX to read the corresponding item from the underlying DynamoDB table. If the item doesn't exist in DynamoDB, DAX stores an empty item in its item cache and returns the empty item to the application. Now suppose that the application sends another `GetItem` request for the same item. DAX finds the empty item in the item cache and returns it to the application immediately. It does not consult DynamoDB at all.

A negative cache entry remains in the DAX item cache until its item TTL has expired, its LRU is invoked, or the item is modified using `PutItem`, `UpdateItem`, or `DeleteItem`.

The DAX query cache handles negative cache results in a similar way. If an application performs a `Query` or `Scan`, and the DAX query cache doesn't contain a cached result, DAX sends the request to DynamoDB. If there are no matching items in the result set, DAX stores an empty result set in the query cache and returns the empty result set to the application. Subsequent `Query` or `Scan` requests yield the same (empty) result set, until the TTL for that result set has expired.

Strategies for writes

The write-through behavior of DAX is appropriate for many application patterns. However, there are some application patterns where a write-through model might not be appropriate.

For applications that are sensitive to latency, writing through DAX incurs an extra network hop. So a write to DAX is a little slower than a write directly to DynamoDB. If your application is sensitive to write latency, you can reduce the latency by writing directly to DynamoDB instead. For more information, see Write-around.

For write-intensive applications (such as those that perform bulk data loading), you might not want to write all of the data through DAX because only a small percentage of that data is ever read by the application. When you write large amounts of data through DAX, it must invoke its LRU algorithm to make room in the cache for the new items to be read. This diminishes the effectiveness of DAX as a read cache.

When you write an item to DAX, the item cache state is altered to accommodate the new item. (For example, DAX might need to evict older data from the item cache to make room for the new item.)

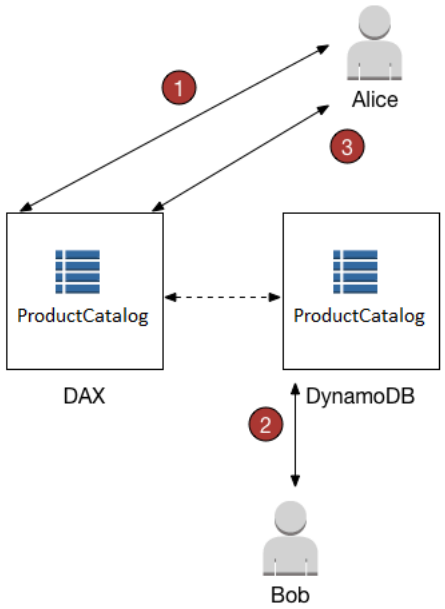
The new item remains in the item cache, subject to the cache's LRU algorithm and the TTL setting for the cache. As long as the item persists in the item cache, DAX doesn't re-read the item from DynamoDB.

Write-through

The DAX item cache implements a write-through policy. For more information, see How DAX processes writes.

When you write an item, DAX ensures that the cached item is synchronized with the item as it exists in DynamoDB. This is helpful for applications that need to re-read an item immediately after writing it. However, if other applications write directly to a DynamoDB table, the item in the DAX item cache is no longer in sync with DynamoDB.

To illustrate, consider two users (Alice and Bob), who are working with the `ProductCatalog` table. Alice accesses the table using DAX, but Bob bypasses DAX and accesses the table directly in DynamoDB.



1. Alice updates an item in the `ProductCatalog` table. DAX forwards the request to DynamoDB, and the update succeeds. DAX then writes the item to its item cache and returns a successful response to Alice. From that point on, until the item is ultimately evicted from the cache, any user who reads the item from DAX sees the item with Alice's update.

2. A short time later, Bob updates the same `ProductCatalog` item that Alice wrote. However, Bob updates the item directly in DynamoDB. DAX does not automatically refresh its item cache in response to updates via DynamoDB. Therefore, DAX users don't see Bob's update.
3. Alice reads the item from DAX again. The item is in the item cache, so DAX returns it to Alice without accessing the DynamoDB table.

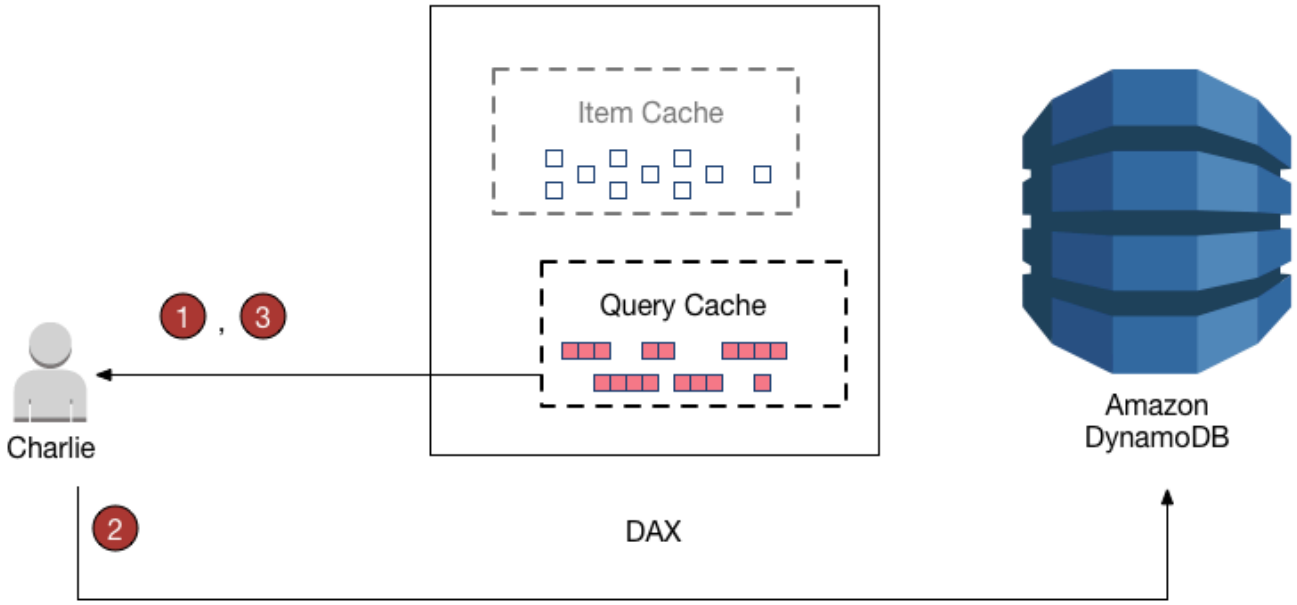
In this scenario, Alice and Bob see different representations of the same `ProductCatalog` item. This is the case until DAX evicts the item from the item cache, or until another user updates the same item again using DAX.

Write-around

If your application needs to write large quantities of data (such as a bulk data load), it might make sense to bypass DAX and write the data directly to DynamoDB. Such a *write-around* strategy reduces write latency. However, the item cache doesn't remain in sync with the data in DynamoDB.

If you decide to use a write-around strategy, remember that DAX populates its item cache whenever applications use the DAX client to read data. This can be advantageous in some cases because it ensures that only the most frequently read data is cached (as opposed to the most frequently written data).

For example, consider a user (Charlie) who wants to work with a different table, the `GameScores` table, using DAX. The partition key for `GameScores` is `UserId`, so all of Charlie's scores would have the same `UserId`.



1. Charlie wants to retrieve all of his scores, so he sends a `Query` to DAX. Assuming that this query has not been issued before, DAX forwards the query to DynamoDB for processing. It stores the results in the DAX query cache, and then returns the results to Charlie. The result set remains available in the query cache until it is evicted.
2. Now suppose that Charlie plays the Meteor Blasters game and achieves a high score. Charlie sends an `UpdateItem` request to DynamoDB, modifying an item in the `GameScores` table.
3. Finally, Charlie decides to rerun his earlier `Query` to retrieve all of his data from `GameScores`. Charlie does not see his high score for Meteor Blasters in the results. This is because the query results come from the query cache, not the item cache. The two caches are independent from one another, so a change in one cache does not affect the other cache.

DAX does not refresh result sets in the query cache with the most current data from DynamoDB. Each result set in the query cache is current as of the time that the `Query` or `Scan` operation was performed. Thus, Charlie's `Query` results don't reflect his `PutItem` operation. This is the case until DAX evicts the result set from the query cache.

Developing with the DynamoDB Accelerator (DAX) client

To use DAX from an application, you use the DAX client for your programming language. The DAX client is designed for minimal disruption to your existing Amazon DynamoDB applications—with only a few simple code modifications needed.

> **Note**
>
> DAX clients for various programming languages are available on the following site:
>
> - http://dax-sdk.s3-website-us-west-2.amazonaws.com

This section demonstrates how to launch an Amazon EC2 instance in your default Amazon VPC, connect to the instance, and run a sample application. It also provides information about how to modify your existing application so that it can use your DAX cluster.

Topics

- Tutorial: Running a sample application using DynamoDB Accelerator (DAX)
- Modifying an existing application to use DAX

Tutorial: Running a sample application using DynamoDB Accelerator (DAX)

This tutorial demonstrates how to launch an Amazon EC2 instance in your default virtual private cloud (VPC), connect to the instance, and run a sample application that uses Amazon DynamoDB Accelerator (DAX).

> **Note**
>
> To complete this tutorial, you must have a DAX cluster running in your default VPC. If you haven't created a DAX cluster, see Creating a DAX cluster for instructions.

Topics

- Step 1: Launch an Amazon EC2 instance

- Step 2: Create a user and policy
- Step 3: Configure an Amazon EC2 instance
- Step 4: Run a sample application

Step 1: Launch an Amazon EC2 instance

When your Amazon DynamoDB Accelerator (DAX) cluster is available, you can launch an Amazon EC2 instance in your default Amazon Virtual Private Cloud (Amazon VPC). You can then install and run DAX client software on that instance.

To launch an EC2 instance

1. Sign in to the AWS Management Console and open the Amazon EC2 console at https://console.aws.amazon.com/ec2/.
2. Choose **Launch Instance**, and do the following:

   Step 1: Choose an Amazon Machine Image (AMI)

   1. In the list of AMIs, find the **Amazon Linux AMI**, and choose **Select**.

   Step 2: Choose an Instance Type

   1. In the list of instance types, choose **t2.micro**.
   2. Choose **Next: Configure Instance Details**.

   Step 3: Configure Instance Details

   1. For **Network**, choose your default VPC.
   2. Choose **Next: Add Storage**.

   Step 4: Add Storage

   1. Skip this step by choosing **Next: Add Tags**.

   Step 5: Add Tags

   1. Skip this step by choosing **Next: Configure Security Group**.

Step 6: Configure Security Group

1. Choose **Select an existing security group**.
2. In the list of security groups, choose **default**. This is the default security group for your VPC.
3. Choose **Next: Review and Launch**.

Step 7: Review Instance Launch

1. Choose **Launch**.

3. In the **Select an existing key pair or create a new key pair** window, do one of the following:
   - If you don't have an Amazon EC2 key pair, choose **Create a new key pair** and follow the instructions. You are asked to download a private key file (`.pem` file). You need this file later when you log in to your Amazon EC2 instance.
   - If you already have an existing Amazon EC2 key pair, go to **Select a key pair** and choose your key pair from the list. You must already have the private key file (`.pem` file) available in order to log in to your Amazon EC2 instance.
4. After configuring your key pair, choose **Launch Instances**.
5. In the console navigation pane, choose **EC2 Dashboard**, and then choose the instance that you launched. In the lower pane, on the **Description** tab, find the **Public DNS** for your instance, for example: `ec2-11-22-33-44.us-west-2.compute.amazonaws.com`. Make a note of this public DNS name because you need it for Step 3: Configure an Amazon EC2 instance.

> **Note**
>
> It takes a few minutes for your Amazon EC2 instance to become available. In the meantime, proceed to Step 2: Create a user and policy and follow the instructions there.

Step 2: Create a user and policy

In this step, you create a user with a policy that grants access to your Amazon DynamoDB Accelerator (DAX) cluster and to DynamoDB using AWS Identity and Access Management. You can then run applications that interact with your DAX cluster.

Sign up for an AWS account

If you do not have an AWS account, complete the following steps to create one.

To sign up for an AWS account

1. Open https://portal.aws.amazon.com/billing/signup.
2. Follow the online instructions.

   Part of the sign-up procedure involves receiving a phone call or text message and entering a verification code on the phone keypad.

   When you sign up for an AWS account, an *AWS account root user* is created. The root user has access to all AWS services and resources in the account. As a security best practice, assign administrative access to a user, and use only the root user to perform tasks that require root user access.

AWS sends you a confirmation email after the sign-up process is complete. At any time, you can view your current account activity and manage your account by going to https://aws.amazon.com/ and choosing **My Account**.

Create a user with administrative access

After you sign up for an AWS account, secure your AWS account root user, enable AWS IAM Identity Center, and create an administrative user so that you don't use the root user for everyday tasks.

Secure your AWS account root user

1. Sign in to the AWS Management Console as the account owner by choosing **Root user** and entering your AWS account email address. On the next page, enter your password.

   For help signing in by using root user, see Signing in as the root user in the *AWS Sign-In User Guide*.
2. Turn on multi-factor authentication (MFA) for your root user.

   For instructions, see Enable a virtual MFA device for your AWS account root user (console) in the *IAM User Guide*.

Create a user with administrative access

1. Enable IAM Identity Center.

   For instructions, see Enabling AWS IAM Identity Center in the *AWS IAM Identity Center User Guide*.

2. In IAM Identity Center, grant administrative access to a user.

   For a tutorial about using the IAM Identity Center directory as your identity source, see Configure user access with the default IAM Identity Center directory in the *AWS IAM Identity Center User Guide*.

Sign in as the user with administrative access

- To sign in with your IAM Identity Center user, use the sign-in URL that was sent to your email address when you created the IAM Identity Center user.

  For help signing in using an IAM Identity Center user, see Signing in to the AWS access portal in the *AWS Sign-In User Guide*.

Assign access to additional users

1. In IAM Identity Center, create a permission set that follows the best practice of applying least-privilege permissions.

   For instructions, see Create a permission set in the *AWS IAM Identity Center User Guide*.

2. Assign users to a group, and then assign single sign-on access to the group.

   For instructions, see Add groups in the *AWS IAM Identity Center User Guide*.

To provide access, add permissions to your users, groups, or roles:

- Users and groups in AWS IAM Identity Center:

  Create a permission set. Follow the instructions in Create a permission set in the *AWS IAM Identity Center User Guide*.

- Users managed in IAM through an identity provider:

Create a role for identity federation. Follow the instructions in Create a role for a third-party identity provider (federation) in the *IAM User Guide*.

- IAM users:
  - Create a role that your user can assume. Follow the instructions in Create a role for an IAM user in the *IAM User Guide*.
  - (Not recommended) Attach a policy directly to a user or add a user to a user group. Follow the instructions in Adding permissions to a user (console) in the *IAM User Guide*.

To use the JSON policy editor to create a policy

1. Sign in to the AWS Management Console and open the IAM console at https://console.aws.amazon.com/iam/.
2. In the navigation pane on the left, choose **Policies**.

   If this is your first time choosing **Policies**, the **Welcome to Managed Policies** page appears. Choose **Get Started**.
3. At the top of the page, choose **Create policy**.
4. In the **Policy editor** section, choose the **JSON** option.
5. Enter or paste a JSON policy document. For details about the IAM policy language, see IAM JSON policy reference.
6. Resolve any security warnings, errors, or general warnings generated during policy validation, and then choose **Next**.

   > **Note**
   >
   > You can switch between the **Visual** and **JSON** editor options anytime. However, if you make changes or choose **Next** in the **Visual** editor, IAM might restructure your policy to optimize it for the visual editor. For more information, see Policy restructuring in the *IAM User Guide*.
7. (Optional) When you create or edit a policy in the AWS Management Console, you can generate a JSON or YAML policy template that you can use in CloudFormation templates.

To do this, in the **Policy editor** choose **Actions**, and then choose **Generate CloudFormation template**. To learn more about CloudFormation, see AWS Identity and Access Management resource type reference in the *AWS CloudFormation User Guide*.

8. When you are finished adding permissions to the policy, choose **Next**.
9. On the **Review and create** page, enter a **Policy name** and a **Description** (optional) for the policy that you are creating. Review **Permissions defined in this policy** to see the permissions that are granted by your policy.
10. (Optional) Add metadata to the policy by attaching tags as key-value pairs. For more information about using tags in IAM, see Tags for AWS Identity and Access Management resources in the *IAM User Guide*.
11. Choose **Create policy** to save your new policy.

Policy document – Copy and paste the following document to create the JSON policy.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Action": [
                "dax:*"
            ],
            "Effect": "Allow",
            "Resource": [
                "*"
            ]
        },
        {
            "Action": [
                "dynamodb:*"
            ],
            "Effect": "Allow",
            "Resource": [
                "*"
            ]
        }
    ]
```

```
}
```

Step 3: Configure an Amazon EC2 instance

When your Amazon EC2 instance is available, you can log in to the instance and prepare it for use.

> **Note**
>
> The following steps assume that you are connecting to your Amazon EC2 instance from a computer running Linux. For other ways to connect, see Connect to Your Linux Instance in the *Amazon EC2 User Guide*.

To configure the EC2 instance

1. Open the Amazon EC2 console at https://console.aws.amazon.com/ec2/.
2. Use the `ssh` command to log in to your Amazon EC2 instance, as shown in the following example.

   ```
   ssh -i my-keypair.pem ec2-user@public-dns-name
   ```

   You need to specify your private key file (`.pem` file) and the public DNS name of your instance. (See Step 1: Launch an Amazon EC2 instance.)

   The login ID is `ec2-user`. No password is required.
3. After you log in to your EC2 instance, configure your AWS credentials as shown following. Enter your AWS access key ID and secret key (from Step 2: Create a user and policy), and set the default Region name to your current Region. (In the following example, the default Region name is `us-west-2`.)

   ```
   aws configure

   AWS Access Key ID [None]: AKIAIOSFODNN7EXAMPLE
   AWS Secret Access Key [None]: wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
   Default region name [None]: us-west-2
   Default output format [None]:
   ```

After launching and configuring your Amazon EC2 instance, you can test the functionality of DAX using one of the available sample applications. For more information, see Step 4: Run a sample application.

Step 4: Run a sample application

To help you test Amazon DynamoDB Accelerator (DAX) functionality, you can run one of the available sample applications on your Amazon EC2 instance.

Topics

- Node.js and DAX
- DAX SDK for Go
- Java and DAX
- .NET and DAX
- Python and DAX

Node.js and DAX

Default client configuration for Node.js

When configuring the DAX JavaScript SDK client, you can customize various parameters to optimize performance, connection handling, and error resilience. The following table outlines the default configuration settings that control how your client interacts with the DAX cluster, including timeout values, retry mechanisms, credential management, and health monitoring options. For more information, see DynamoDBClient Operations.

DAX JS SDK client defaults

Parameter	Type	Description
`region` optional	`string`	The AWS Region to use for the DAX client (example - 'us-east-1'). This is a required parameter if not provided through the environment variable.
`endpoint`	`string`	The endpoint of the Cluster to which the SDK connects.

Parameter	Type	Description
required		Examples: Non-encrypted – dax-cluster-name.region.amazonaws.com Encrypted – daxs://my-cluster.l6fzcv.dax-clusters.us-east-1.amazonaws.com
`requestTimeout` default 6000 ms	`number`	This defines the maximum time the client will wait for a response from DAX.
`writeRetries` default 1	`number`	The number of retries to attempt for write requests that fail.
`readRetries` default 1	`number`	The number of retries to attempt for read requests that fail.
`maxRetries` default 1	`number`	The maximum number of retries to attempt on failed requests. If readRetries/writeRetries are set, then the configura tion set in readRetries and writeRetries take priority over maxRetries.
`connectTimeout` default 10000 ms	`number`	The timeout (in milliseconds) for establishing a connection to any of the cluster nodes.

Parameter	Type	Description
`maxConcurrentConnections` default 100	`number`	Limits the total number of concurrent connections that a client instance can create per node in a DAX cluster.
`maxRetryDelay` default 7000 ms	`number`	When the DAX server indicates recover is needed by setting `waitForRecoveryBeforeRetrying` flag to true, the client will pause before retry attempts. During these recovery periods, the `maxRetryDelay` parameter determines the maximum waiting time between retries. This recovery-specific configuration only applies when the DAX server is in recovery mode. For all other scenarios, retry behavior follows one of two patterns: either an exponential delay based on the retry count (governed by `writeRetries`, `readRetries`, or `maxRetries` parameters), or an immediate retry depending on the exception type.

Parameter	Type	Description
`credentials` optional	AwsCredentialIdentity | AwsCredentialIdentityProvider	The AWS credentials to use for authenticating requests. This can be provided as an AwsCredentialIdentity or an AwsCredentialIdentityProvider. If not provided, the AWS SDK will automatically use the default credentials provider chain. Example: `{ accessKeyId: 'AKIA...', secretAccessKey: '...', sessionToken: '...' }` * @default Uses default AWS credentials provider chain.
`healthCheckInterval` default 5000 ms	`number`	The interval (in milliseconds) between cluster health checks. A lower interval will check more frequently.
`healthCheckTimeout` default 1000 ms	`number`	The timeout (in milliseconds) for the health check to complete.

Parameter	Type	Description
`skipHostnameVerification` default false	`boolean`	Skip hostname verification of TLS connections. This has no impact on un-encrypted clusters. The default is to perform hostname verification, setting this to True will skip verification. Be sure you understand the implication of turning it off, which is the inability to authenticate the cluster that you are connecting to.
`unhealthyConsecutiveErrorCount` default 5	`number`	Sets the number of consecutive errors required to signal node unhealthy within health check interval.
`clusterUpdateInterval` default 4000 ms	`number`	Returns the interval between polling of cluster members for membership changes.
`clusterUpdateThreshold` default 125	`number`	Returns the threshold below which the cluster will not be polled for membership changes.
`credentailProvider` optional | default null	`AwsCredentialIdentityProvider`	User Defined Provider for AWS credentials used to authenticate requests to DAX.

Pagination configuration for DaxDocument

Name	Type	Detail
`client`	DaxDocument	Instance of DaxDocument type.
`pageSize`	number	Determines the number of items per page.
`startingToken` Optional	any	LastEvaluatedKey from previous response can be used for subsequent requests.

For usage of pagination, see the section called “TryDax.js”.

Migrating to DAX Node.js SDK V3

This migration guide will help you transition your existing DAX Node.js applications. The new SDK requires Node.js 18 or higher and introduces several important changes in how you'll structure your DynamoDB Accelerator code. This guide will walk you through the key differences, including syntax changes, new import methods, and updated asynchronous programming patterns.

V2 Node.js DAX usage

```
const AmazonDaxClient = require('amazon-dax-client');
const AWS = require('aws-sdk');

var region = "us-west-2";

AWS.config.update({
  region: region,
});

var client = new AWS.DynamoDB.DocumentClient();

if (process.argv.length > 2) {
  var dax = new AmazonDaxClient({
    endpoints: [process.argv[2]],
    region: region,
  });
```

```
  client = new AWS.DynamoDB.DocumentClient({ service: dax });
}

// Make Get Call using Dax
var params = {
    TableName: 'TryDaxTable',
    pk: 1,
    sk: 1
}
client.get(params, function (err, data) {
    if (err) {
        console.error(
            "Unable to read item. Error JSON:",
            JSON.stringify(err, null, 2)
          );
    } else {
        console.log(data);
    }
});
```

V3 Node.js DAX usage

For Using DAX Node.js V3 Node version 18 or above is the preferred version. To move to Node 18, use the following:

```
import { DaxDocument } from '@amazon-dax-sdk/lib-dax';
import { DynamoDBDocument } from '@aws-sdk/lib-dynamodb';
import { DynamoDBClient } from '@aws-sdk/client-dynamodb';

let client: DynamoDBDocument | DaxDocument = DynamoDBDocument.from(
  new DynamoDBClient({ region: 'us-west-2' })
);

if (process.argv.length > 2) {
  client = new DaxDocument({
    endpoints: [process.argv[2]],
    region: 'us-west-2',
  });
}

const params = {
  TableName: 'TryDaxTable',
  Key: { pk: 1, sk: 1 },
```

```
};

try {
  const results = await client.get(params);
  console.log(results);
} catch (err) {
  console.error(err);
}
```

The DAX SDK for Node.js v3.x is compatible with AWS SDK for Node.js v3.x. The DAX SDK for Node.js v3.x supports the use of aggregated clients. Please note that DAX doesn't support the creation of bare-bones clients. For more details on unsupported features, see the section called "Features not in parity with AWS SDK V3".

Follow these steps to run the Node.js sample application on your Amazon EC2 instance.

To run the Node.js sample for DAX

1. Set up Node.js on your Amazon EC2 instance, as follows:

   a. Install node version manager (nvm).

   ```
   curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.40.1/install.sh | bash
   ```

   b. Use nvm to install Node.js.

   ```
   nvm install 18
   ```

   c. Use nvm to use Node 18

   ```
   nvm use 18
   ```

   d. Test that Node.js is installed and running correctly.

   ```
   node -e "console.log('Running Node.js ' + process.version)"
   ```

   This should display the following message.

   ```
   Running Node.js v18.x.x
   ```

2. Install the DaxDocument Node.js client using the node package manager (npm).

```
npm install @amazon-dax-sdk/lib-dax
```

TryDax sample code

To evaluate the performance benefits of DynamoDB Accelerator (DAX), follow these steps to run a sample test that compares read operation times between standard DynamoDB and a DAX cluster.

1. After you've set up your workspace and installed the `lib-dax` as a dependency, copy the section called “TryDax.js” into your project.
2. Run the program against your DAX cluster. To determine the endpoint for your DAX cluster, choose one of the following:

   - **Using the DynamoDB console** — Choose your DAX cluster. The cluster endpoint is shown on the console, as in the following example.

     ```
     dax://my-cluster.l6fzcv.dax-clusters.us-east-1.amazonaws.com
     ```

   - **Using the AWS CLI** — Enter the following command.

     ```
     aws dax describe-clusters --query "Clusters[*].ClusterDiscoveryEndpoint"
     ```

     The cluster endpoint is shown in the output, as in the following example.

     ```
     {
         "Address": "my-cluster.l6fzcv.dax-clusters.us-east-1.amazonaws.com",
         "Port": 8111,
         "URL": "dax://my-cluster.l6fzcv.dax-clusters.us-east-1.amazonaws.com"
     }
     ```

3. Now run the program by specifying the cluster endpoint as a command line parameter.

   ```
   node TryDax.js dax://my-cluster.l6fzcv.dax-clusters.us-east-1.amazonaws.com
   ```

   You should see output similar to the following:

   ```
   Attempting to create table; please wait...
   Successfully created table. Table status: ACTIVE
   ```

```
Writing data to the table...
Writing 20 items for partition key:  1
Writing 20 items for partition key:  2
Writing 20 items for partition key:  3
...
Running GetItem Test
        Total time: 153555.10 µs - Avg time: 383.89 µs
        Total time: 44679.96 µs - Avg time: 111.70 µs
        Total time: 36885.86 µs - Avg time: 92.21 µs
        Total time: 32467.25 µs - Avg time: 81.17 µs
        Total time: 32202.60 µs - Avg time: 80.51 µs
Running Query Test
        Total time: 14869.25 µs - Avg time: 2973.85 µs
        Total time: 3036.31 µs - Avg time: 607.26 µs
        Total time: 2468.92 µs - Avg time: 493.78 µs
        Total time: 2062.53 µs - Avg time: 412.51 µs
        Total time: 2178.22 µs - Avg time: 435.64 µs
Running Scan Test
        Total time: 2395.88 µs - Avg time: 479.18 µs
        Total time: 2207.16 µs - Avg time: 441.43 µs
        Total time: 2443.14 µs - Avg time: 488.63 µs
        Total time: 2038.24 µs - Avg time: 407.65 µs
        Total time: 1972.17 µs - Avg time: 394.43 µs
Running Pagination Test
Scan Pagination
[
  { pk: 1, sk: 1, someData: 'XXXXXXXXXX' },
  { pk: 1, sk: 2, someData: 'XXXXXXXXXX' },
  { pk: 1, sk: 3, someData: 'XXXXXXXXXX' }
]
[
  { pk: 1, sk: 4, someData: 'XXXXXXXXXX' },
  { pk: 1, sk: 5, someData: 'XXXXXXXXXX' },
  { pk: 1, sk: 6, someData: 'XXXXXXXXXX' }
]
...
Query Pagination
[
  { pk: 1, sk: 1, someData: 'XXXXXXXXXX' },
  { pk: 1, sk: 2, someData: 'XXXXXXXXXX' },
  { pk: 1, sk: 3, someData: 'XXXXXXXXXX' }
]
[
  { pk: 1, sk: 4, someData: 'XXXXXXXXXX' },
```

```
  { pk: 1, sk: 5, someData: 'XXXXXXXXXX' },
  { pk: 1, sk: 6, someData: 'XXXXXXXXXX' }
]
...
Attempting to delete table; please wait...
Successfully deleted table.
```

Take note of the timing information. The number of microseconds required for the `GetItem`,`Query`,`Scan` tests.

4. In this case, you ran the programs against the DAX cluster. Now, you'll run the program again, this time against DynamoDB.
5. Now run the program again, but this time, without the cluster endpoint URL as a command line parameter.

```
node TryDax.js
```

Look at the output and take note of the timing information. The elapsed times for `GetItem`, `Query`, and `Scan` should be significantly lower with DAX as compared to DynamoDB.

Features not in parity with AWS SDK V3

- Bare-bones clients – Dax Node.js V3 doesn't support bare-bones clients.

```
const dynamoDBClient = new DynamoDBClient({ region: 'us-west-2' });
const regularParams = {
    TableName: 'TryDaxTable',
    Key: {
        pk: 1,
        sk: 1
    }
};
// The DynamoDB client supports the send operation.
const dynamoResult = await dynamoDBClient.send(new GetCommand(regularParams));

// However, the DaxDocument client does not support the send operation.
const daxClient = new DaxDocument({
    endpoints: ['your-dax-endpoint'],
    region: 'us-west-2',
});
```

```
const params = {
    TableName: 'TryDaxTable',
    Key: {
        pk: 1,
        sk: 1
    }
};

// This will throw an error - send operation is not supported for DAX. Please refer
 to documentation.
const result = await daxClient.send(new GetCommand(params));
console.log(result);
```

- Middleware Stack – Dax Node.js V3 doesn't support using Middleware functions.

```
const dynamoDBClient = new DynamoDBClient({ region: 'us-west-2' });
// The DynamoDB client supports the middlewareStack.
dynamoDBClient.middlewareStack.add(
  (next, context) =>> async (args) => {
    console.log("Before operation:", args);
    const result = await next(args);
    console.log("After operation:", result);
    return result;
  },
  {
    step: "initialize", // or "build", "finalizeRequest", "deserialize"
    name: "loggingMiddleware",
  }
);

// However, the DaxDocument client does not support the middlewareStack.
const daxClient = new DaxDocument({
    endpoints: ['your-dax-endpoint'],
    region: 'us-west-2',
});

// This will throw an error - custom middleware and middlewareStacks are not
 supported for DAX. Please refer to documentation.
daxClient.middlewareStack.add(
  (next, context) => async (args) => {
    console.log("Before operation:", args);
    const result = await next(args);
```

```
    console.log("After operation:", result);
    return result;
  },
  {
    step: "initialize", // or "build", "finalizeRequest", "deserialize"
    name: "loggingMiddleware",
  }
);
```

TryDax.js

```
import { DynamoDB, waitUntilTableExists, waitUntilTableNotExists } from "@aws-sdk/
client-dynamodb";
import { DaxDocument, daxPaginateScan, daxPaginateQuery } from "@amazon-dax-sdk/lib-
dax";
import { DynamoDBDocument, paginateQuery, paginateScan } from "@aws-sdk/lib-dynamodb";

const region = "us-east-1";
const tableName = "TryDaxTable";

// Determine the client (DynamoDB or DAX)
let client = DynamoDBDocument.from(new DynamoDB({ region }));
if (process.argv.length > 2) {
  client = new DaxDocument({ region, endpoint: process.argv[2] });
}

// Function to create table
async function createTable() {
  const dynamodb = new DynamoDB({ region });
  const params = {
    TableName: tableName,
    KeySchema: [
      { AttributeName: "pk", KeyType: "HASH" },
      { AttributeName: "sk", KeyType: "RANGE" },
    ],
    AttributeDefinitions: [
      { AttributeName: "pk", AttributeType: "N" },
      { AttributeName: "sk", AttributeType: "N" },
    ],
    ProvisionedThroughput: { ReadCapacityUnits: 25, WriteCapacityUnits: 25 },
  };
```

```
  try {
    console.log("Attempting to create table; please wait...");
    await dynamodb.createTable(params);
    await waitUntilTableExists({ client: dynamodb, maxWaitTime: 30 }, { TableName:
 tableName });
    console.log("Successfully created table. Table status: ACTIVE");
  } catch (err) {
    console.error("Error in table creation:", err);
  }
}

// Function to insert data
async function writeData() {
  console.log("Writing data to the table...");
  const someData = "X".repeat(10);
  for (let ipk = 1; ipk <= 20; ipk++) {
    console.log("Writing 20 items for partition key: ", ipk)
    for (let isk = 1; isk <= 20; isk++) {
      try {
        await client.put({ TableName: tableName, Item: { pk: ipk, sk: isk,
 someData } });
      } catch (err) {
        console.error("Error inserting data:", err);
      }
    }
  }
}

// Function to test GetItem
async function getItemTest() {
  console.log("Running GetItem Test");
  for (let i = 0; i < 5; i++) {
    const startTime = performance.now();
    const promises = [];
    for (let ipk = 1; ipk <= 20; ipk++) {
      for (let isk = 1; isk <= 20; isk++) {
        promises.push(client.get({ TableName: tableName, Key: { pk: ipk, sk: isk } }));
      }
    }
    await Promise.all(promises);
    const endTime = performance.now();
    const iterTime = (endTime - startTime) * 1000;
    const totalTime = iterTime.toFixed(2).padStart(3, ' ');
```

```
    const avgTime = (iterTime / 400).toFixed(2).padStart(3, ' ');
    console.log(`\tTotal time: ${totalTime} \u00B5s - Avg time: ${avgTime} \u00B5s`);
  }
}

// Function to test Query
async function queryTest() {
  console.log("Running Query Test");
  for (let i = 0; i < 5; i++) {
    const startTime = performance.now();
    const promises = [];
    for (let pk = 1; pk <= 5; pk++) {
      const params = {
        TableName: tableName,
        KeyConditionExpression: "pk = :pkval and sk between :skval1 and :skval2",
        ExpressionAttributeValues: { ":pkval": pk, ":skval1": 1, ":skval2": 2 },
      };
      promises.push(client.query(params));
    }
    await Promise.all(promises);
    const endTime = performance.now();
    const iterTime = (endTime - startTime) * 1000;
    const totalTime = iterTime.toFixed(2).padStart(3, ' ');
    const avgTime = (iterTime / 5).toFixed(2).padStart(3, ' ');
    console.log(`\tTotal time: ${totalTime} \u00B5s - Avg time: ${avgTime} \u00B5s`);
  }
}

// Function to test Scan
async function scanTest() {
  console.log("Running Scan Test");
  for (let i = 0; i < 5; i++) {
    const startTime = performance.now();
    const promises = [];
    for (let pk = 1; pk <= 5; pk++) {
      const params = {
        TableName: tableName,
        FilterExpression: "pk = :pkval and sk between :skval1 and :skval2",
        ExpressionAttributeValues: { ":pkval": pk, ":skval1": 1, ":skval2": 2 },
      };
      promises.push(client.scan(params));
    }
    await Promise.all(promises);
    const endTime = performance.now();
```

```
    const iterTime = (endTime - startTime) * 1000;
    const totalTime = iterTime.toFixed(2).padStart(3, ' ');
    const avgTime = (iterTime / 5).toFixed(2).padStart(3, ' ');
    console.log(`\tTotal time: ${totalTime} \u00B5s - Avg time: ${avgTime} \u00B5s`);
  }
}

// Function to test Pagination
async function paginationTest() {
  console.log("Running Pagination Test");
  console.log("Scan Pagination");
  const scanParams = { TableName: tableName };
  const paginator = process.argv.length > 2 ? daxPaginateScan : paginateScan;
  for await (const page of paginator({ client, pageSize: 3 }, scanParams)) {
    console.log(page.Items);
  }

  console.log("Query Pagination");
  const queryParams = {
    TableName: tableName,
    KeyConditionExpression: "pk = :pkval and sk between :skval1 and :skval2",
    ExpressionAttributeValues: { ":pkval": 1, ":skval1": 1, ":skval2": 10 },
  };
  const queryPaginator = process.argv.length > 2 ? daxPaginateQuery : paginateQuery;
  for await (const page of queryPaginator({ client, pageSize: 3 }, queryParams)) {
    console.log(page.Items);
  }
}

// Function to delete the table
async function deleteTable() {
  const dynamodb = new DynamoDB({ region });
  console.log("Attempting to delete table; please wait...")
  try {
    await dynamodb.deleteTable({ TableName: tableName });
    await waitUntilTableNotExists({ client: dynamodb, maxWaitTime: 30 }, { TableName:
 tableName });
    console.log("Successfully deleted table.");
  } catch (err) {
    console.error("Error deleting table:", err);
  }
}

// Execute functions sequentially
```

```
(async function () {
  await createTable();
  await writeData();
  await getItemTest();
  await queryTest();
  await scanTest();
  await paginationTest();
  await deleteTable();
})();
```

DAX SDK for Go

Follow this procedure to run the Amazon DynamoDB Accelerator (DAX) SDK for Go sample application on your Amazon EC2 instance.

To run the SDK for Go sample for DAX

1. Set up the SDK for Go on your Amazon EC2 instance:

   a. Install the Go programming language (`Golang`).

   ```
   sudo yum install -y golang
   ```

   b. Test that Golang is installed and running correctly.

   ```
   go version
   ```

   A message like this should appear.

   ```
   go version go1.23.4 linux/amd64
   ```

2. Install the sample Golang application.

   ```
   go get github.com/aws-samples/sample-aws-dax-go-v2
   ```

3. Run the following Golang programs. The first program creates a DynamoDB table named `TryDaxGoTable`. The second program writes data to the table.

   ```
   go run ~/go/pkg/mod/github.com/aws-samples/sample-aws-dax-go-v2@v1.0.0/try_dax.go -
   service dynamodb -command create-table
   ```

```
go run ~/go/pkg/mod/github.com/aws-samples/sample-aws-dax-go-v2@v1.0.0/try_dax.go -
service dynamodb -command put-item
```

4. Run the following Golang programs.

```
go run ~/go/pkg/mod/github.com/aws-samples/sample-aws-dax-go-v2@v1.0.0/try_dax.go -
service dynamodb -command get-item
```

```
go run ~/go/pkg/mod/github.com/aws-samples/sample-aws-dax-go-v2@v1.0.0/try_dax.go -
service dynamodb -command query
```

```
go run ~/go/pkg/mod/github.com/aws-samples/sample-aws-dax-go-v2@v1.0.0/try_dax.go -
service dynamodb -command scan
```

   Take note of the timing information—the number of milliseconds required for the `GetItem`, `Query`, and `Scan` tests.

5. In the previous step, you ran the programs against the DynamoDB endpoint. Now, run the programs again, but this time, the `GetItem`, `Query`, and `Scan` operations are processed by your DAX cluster.

   To determine the endpoint for your DAX cluster, choose one of the following:

   - **Using the DynamoDB console** — Choose your DAX cluster. The cluster endpoint is shown on the console, as in the following example.

     ```
     dax://my-cluster.l6fzcv.dax-clusters.us-east-1.amazonaws.com
     ```

   - **Using the AWS CLI** — Enter the following command.

     ```
     aws dax describe-clusters --query "Clusters[*].ClusterDiscoveryEndpoint"
     ```

     The cluster endpoint is shown in the output, as in the following example.

     ```
     {
         "Address": "my-cluster.l6fzcv.dax-clusters.us-east-1.amazonaws.com",
         "Port": 8111,
         "URL": "dax://my-cluster.l6fzcv.dax-clusters.us-east-1.amazonaws.com"
     ```

```
}
```

Now run the programs again, but this time, specify the cluster endpoint as a command line parameter.

```
go run ~/go/pkg/mod/github.com/aws-samples/sample-aws-dax-go-v2@v1.0.0/try_dax.go
 -service dax -command get-item -endpoint my-cluster.l6fzcv.dax-clusters.us-
east-1.amazonaws.com:8111
```

```
go run ~/go/pkg/mod/github.com/aws-samples/sample-aws-dax-go-v2@v1.0.0/try_dax.go
 -service dax -command query -endpoint my-cluster.l6fzcv.dax-clusters.us-
east-1.amazonaws.com:8111
```

```
go run ~/go/pkg/mod/github.com/aws-samples/sample-aws-dax-go-v2@v1.0.0/try_dax.go
 -service dax -command scan -endpoint my-cluster.l6fzcv.dax-clusters.us-
east-1.amazonaws.com:8111
```

```
go run ~/go/pkg/mod/github.com/aws-samples/sample-aws-dax-go-v2@v1.0.0/try_dax.go
 -service dax -command paginated-scan -endpoint my-cluster.l6fzcv.dax-clusters.us-
east-1.amazonaws.com:8111
```

```
go run ~/go/pkg/mod/github.com/aws-samples/sample-aws-dax-go-v2@v1.0.0/try_dax.go
 -service dax -command paginated-query -endpoint my-cluster.l6fzcv.dax-clusters.us-
east-1.amazonaws.com:8111
```

```
go run ~/go/pkg/mod/github.com/aws-samples/sample-aws-dax-go-v2@v1.0.0/try_dax.go
 -service dax -command paginated-batch-get -endpoint my-cluster.l6fzcv.dax-
clusters.us-east-1.amazonaws.com:8111
```

Look at the rest of the output, and take note of the timing information. The elapsed times for `GetItem`, `Query`, and `Scan` should be significantly lower with DAX than with DynamoDB.

6. Run the following Golang program to delete `TryDaxGoTable`.

```
go run ~/go/pkg/mod/github.com/aws-samples/sample-aws-dax-go-v2@v1.0.0/try_dax.go -
service dynamodb -command delete-table
```

Features not in parity with AWS SDK for Go V2

Middleware Stack – DAX Go V2 doesn't support the use of Middleware Stacks through APIoptions. For more information, see Customizing the AWS SDK for Go v2 Client Requests with Middleware.

Example:

```
// Custom middleware implementation
type customSerializeMiddleware struct{}
// ID returns the identifier for the middleware
func (m *customSerializeMiddleware) ID() string {
    return "CustomMiddleware"
}
// HandleSerialize implements the serialize middleware handler
func (m *customSerializeMiddleware) HandleSerialize(
    ctx context.Context,
    in middleware.SerializeInput,
    next middleware.SerializeHandler,
) (
    out middleware.SerializeOutput,
    metadata middleware.Metadata,
    err error,
) {
    // Add your custom logic here before the request is serialized
    fmt.Printf("Executing custom middleware for request: %v\n", in)
    // Call the next handler in the middleware chain
    return next.HandleSerialize(ctx, in)
}

func executeGetItem(ctx context.Context) error {
    client, err := initItemClient(ctx)
    if err != nil {
        os.Stderr.WriteString(fmt.Sprintf("failed to initialize client: %v\n", err))
        return err
    }

    st := time.Now()
    for c := 0; c < iterations; c++ {
        for i := 0; i < pkMax; i++ {
            for j := 0; j < skMax; j++ {
                // Create key using attributevalue.Marshal for type safety
                pk, err := attributevalue.Marshal(fmt.Sprintf("%s_%d", keyPrefix, i))
                if err != nil {
                    return fmt.Errorf("error marshaling pk: %v", err)
```

```
                }
                sk, err := attributevalue.Marshal(fmt.Sprintf("%d", j))
                if err != nil {
                    return fmt.Errorf("error marshaling sk: %v", err)
                }
                key := map[string]types.AttributeValue{
                    "pk": pk,
                    "sk": sk,
                }
                in := &dynamodb.GetItemInput{
                    TableName: aws.String(table),
                    Key:       key,
                }

                // Custom middleware option
                customMiddleware := func(o *dynamodb.Options) {
                    o.APIOptions = append(o.APIOptions, func(stack *middleware.Stack)
 error {
                        // Add custom middleware to the stack
                        return stack.Serialize.Add(&customSerializeMiddleware{},
 middleware.After)
                    })
                }

                // Apply options to the GetItem call
                out, err := client.GetItem(ctx, in, customMiddleware)
                if err != nil {
                    return err
                }
                writeVerbose(out)
            }
        }
    }
    d := time.Since(st)
    os.Stdout.WriteString(fmt.Sprintf("Total Time: %v, Avg Time: %v\n", d, d/
iterations))
    return nil
}
```

Output:

```
failed to execute command: custom middleware through APIOptions is not supported in DAX
 client
```

```
exit status 1
```

Default client configuration for Go

This guide will walk you through the configuration options that allow you to fine-tune your DAX client's performance, connection management, and logging behavior. By understanding the default settings and how to customize them, you can optimize your Go application's interaction with DAX.

In this section

- DAX Go SDK Client Defaults
- Client creation

DAX Go SDK Client Defaults

Parameter	Type	Description
`Region` required	`string`	The AWS Region to use for the DAX client (example- 'us-east-1'). This is a required parameter if not provided through the environment.
`HostPorts` required	`[] string`	List of DAX cluster endpoints to which SDK connects. For example: Non-Encrypted – dax://my-cluster.l6fzcv.dax-clusters.us-east-1.amazonaws.com Encrypted – daxs://my-cluster.l6fzcv.dax-clusters.us-east-1.amazonaws.com
`MaxPendingConnectionsPerHost` default 10	`number`	Number of concurrent connection attempts. (Connections can be in the

Parameter	Type	Description
		process of being established concurrently.)
`ClusterUpdateThreshold` default 125 * time.Millisecond	`time.Duration`	The minimum time that must elapse between cluster refreshes.
`ClusterUpdateInterval` default 4 * time.Second	`time.Duration`	The interval at which the client will automatically refresh the DAX cluster information.
`IdleConnectionsReapDelay` default 30 * time.Second	`time.Duration`	The interval at which the client will close idle connections in the DAX client.
`ClientHealthCheckInterval` default 5 * time.Second	`time.Duration`	The interval at which the client will perform health checks on the DAX cluster endpoints.
`Credentials` default	`aws.CredentialsProvider`	The AWS credentials used by the DAX client to authenticate requests to the DAX service. See Credentials and Credential Providers.
`DialContext` default	`func`	A custom function used by the DAX client to establish connections to the DAX cluster.

Parameter	Type	Description
`SkipHostnameVerification` default false	bool	Skip hostname verification of TLS connections. This setting only affects encrypted clusters. When set to True, it disables hostname verification. Disabling verification means you can't authenticate the identity of the cluster you're connecting to, which poses security risks. By default, hostname verification is enabled.
`RouteManagerEnabled` default false	`bool`	This flag is used to remove routes facing network errors.
`RequestTimeout` default 60 * time.Second	`time.Duration`	This defines the maximum time the client will wait for a response from DAX. Priority: Context timeout (if set) > `RequestTimmeout` (if set) > Default 60s `RequestTimeout` .
`WriteRetries` default 2	`number`	The number of retries to attempt for write requests that fail.
`ReadRetries` default 2	`number`	The number of retries to attempt for read requests that fail.

Parameter	Type	Description
`RetryDelay` default 0	`time.Duration`	The delay for non-throttled errors (in seconds) for retry attempts when a request fails.
`Logger` optional	`logging.Logger`	Logger is an interface for logging entries at certain classifications.
`LogLevel` default utils.LogOff	`number`	This loglevel is defined for DAX only. It can be imported using github.com/aws/aws-dax-go-v2/tree/main/dax/utils. `const ( LogOff LogLevelType = 0 LogDebug LogLevelType = 1 LogDebugWithRequestRetries LogLevelType = 2 )`

Note

For `time.Duration`, the default unit is nanosecond. If we don't specify any unit for any parameter then it will consider that as nano seconds: `daxCfg.ClusterUpdateInterval = 10` means 10 nano seconds. (`daxCfg.ClusterUpdateInterval = 10 * time.Millisecond` means 10 milliseconds).

Client creation

To create a DAX client:

- Create DAX config, then create DAX client using DAX config. Using this, you can overwrite a DAX configuration if required.

```
import (
"github.com/aws/aws-dax-go-v2/dax/utils"
"github.com/aws/aws-dax-go-v2/dax"
)

// Non - Encrypted : 'dax://my-cluster.l6fzcv.dax-clusters.us-east-1.amazonaws.com'.
// Encrypted : daxs://my-cluster.l6fzcv.dax-clusters.us-east-1.amazonaws.com'.

config := dax.DefaultConfig()
config.HostPorts = []string{endpoint}
config.Region = region
config.LogLevel = utils.LogDebug
daxClient, err := dax.New(config)
```

Migrating to DAX Go SDK V2

This migration guide will help you transition your existing DAX Go applications.

V1 DAX Go SDK usage

```
package main

import (
    "fmt"
    "os"

    "github.com/aws/aws-dax-go/dax"
    "github.com/aws/aws-sdk-go/aws"
    "github.com/aws/aws-sdk-go/aws/session"
    "github.com/aws/aws-sdk-go/service/dynamodb"
)

func main() {
    region := "us-west-2"
    endpoint := "dax://my-cluster.l6fzcv.dax-clusters.us-east-1.amazonaws.com"
```

```
    // Create session
    sess, err := session.NewSession(&aws.Config{
        Region: aws.String(region),
    })
    if err != nil {
        fmt.Printf("Failed to create session: %v\n", err)
        os.Exit(1)
    }

    // Configure DAX client
    cfg := dax.DefaultConfig()
    cfg.HostPorts = []string{endpoint}
    cfg.Region = region

    // Create DAX client
    daxClient, err := dax.New(cfg)
    if err != nil {
        fmt.Printf("Failed to create DAX client: %v\n", err)
        os.Exit(1)
    }
    defer daxClient.Close() // Don't forget to close the client

    // Create GetItem input
    input := &dynamodb.GetItemInput{
        TableName: aws.String("TryDaxTable"),
        Key: map[string]*dynamodb.AttributeValue{
            "pk": {
                N: aws.String("1"),
            },
            "sk": {
                N: aws.String("1"),
            },
        },
    }

    // Make the GetItem call
    result, err := daxClient.GetItem(input)
    if err != nil {
        fmt.Printf("Failed to get item: %v\n", err)
        os.Exit(1)
    }

    // Print the result
```

```
    fmt.Printf("GetItem succeeded: %+v\n", result)
}
```

V2 DAX Go SDK usage

```
package main

import (
    "context"
    "fmt"
    "os"

    "github.com/aws/aws-dax-go-v2/dax"
    "github.com/aws/aws-sdk-go-v2/config"
    "github.com/aws/aws-sdk-go-v2/service/dynamodb"
    "github.com/aws/aws-sdk-go-v2/service/dynamodb/types"
    "github.com/aws/aws-sdk-go-v2/feature/dynamodb/attributevalue"
    "github.com/aws/aws-sdk-go-v2/aws"
)

func main() {
    ctx := context.Background()
    region := "us-west-2"
    endpoint := "dax://my-cluster.l6fzcv.dax-clusters.us-east-1.amazonaws.com"

    // Create DAX config
    config := dax.DefaultConfig()
    // Specify Endpoint and Region
    config.HostPorts = []string{endpoint}
    config.Region = region
    // Enabling logging
    config.LogLevel = utils.LogDebug
    // Create DAX client
    daxClient, err := dax.New(config)
    if err != nil {
        fmt.Printf("Failed to create DAX client: %v\n", err)
        os.Exit(1)
    }
    defer daxClient.Close() // Don't forget to close the client

    // Create key using attributevalue.Marshal for type safety
    pk, err := attributevalue.Marshal(fmt.Sprintf("%s_%d", keyPrefix, i))
    if err != nil {
```

```
        return fmt.Errorf("error marshaling pk: %v", err)
    }
    sk, err := attributevalue.Marshal(fmt.Sprintf("%d", j))
    if err != nil {
        return fmt.Errorf("error marshaling sk: %v", err)
    }

    // Create GetItem input
    input := &dynamodb.GetItemInput{
        TableName: aws.String("TryDaxTable"),
        Key: map[string]types.AttributeValue{
             "pk": pk,
             "sk": sk,
        },
    }

    // Make the GetItem call
    result, err := daxClient.GetItem(ctx, input)
    if err != nil {
        fmt.Printf("Failed to get item: %v\n", err)
        os.Exit(1)
    }

    // Print the result
    fmt.Printf("GetItem succeeded: %+v\n", result)
}
```

For more API usage details, see AWS samples.

Java and DAX

DAX SDK for Java 2.x is compatible with AWS SDK for Java 2.x. It's built on top of Java 8+ and includes support for non-blocking I/O. For information about using DAX with AWS SDK for Java 1.x, see Using DAX with AWS SDK for Java 1.x.

Using the client as a Maven dependency

Follow these steps to use the client for the DAX SDK for Java in your application as a dependency.

1. Download and install Apache Maven. For more information, see Downloading Apache Maven and Installing Apache Maven.
2. Add the client Maven dependency to your application's Project Object Model (POM) file. In this example, replace *x.x.x* with the actual version number of the client.

```
<!--Dependency:-->
<dependencies>
    <dependency>
        <groupId>software.amazon.dax</groupId>
        <artifactId>amazon-dax-client</artifactId>
        <version>x.x.x</version>
    </dependency>
</dependencies>
```

TryDax sample code

After you've set up your workspace and added the DAX SDK as a dependency, copy TryDax.java into your project.

Run the code using this command.

```
java -cp classpath TryDax
```

You should see output similar to the following.

```
Creating a DynamoDB client

Attempting to create table; please wait...
Successfully created table.  Table status: ACTIVE
Writing data to the table...
Writing 10 items for partition key: 1
Writing 10 items for partition key: 2
Writing 10 items for partition key: 3
...

Running GetItem and Query tests...
First iteration of each test will result in cache misses
Next iterations are cache hits

GetItem test - partition key 1-100 and sort keys 1-10
  Total time: 4390.240 ms - Avg time: 4.390 ms
  Total time: 3097.089 ms - Avg time: 3.097 ms
  Total time: 3273.463 ms - Avg time: 3.273 ms
  Total time: 3353.739 ms - Avg time: 3.354 ms
  Total time: 3533.314 ms - Avg time: 3.533 ms
```

```
Query test - partition key 1-100 and sort keys between 2 and 9
  Total time: 475.868 ms - Avg time: 4.759 ms
  Total time: 423.333 ms - Avg time: 4.233 ms
  Total time: 460.271 ms - Avg time: 4.603 ms
  Total time: 397.859 ms - Avg time: 3.979 ms
  Total time: 466.644 ms - Avg time: 4.666 ms

Attempting to delete table; please wait...
Successfully deleted table.
```

Take note of the timing information—the number of milliseconds required for the `GetItem` and `Query` tests. In this case, you ran the program against the DynamoDB endpoint. Now you'll run the program again, this time against your DAX cluster.

To determine the endpoint of your DAX cluster, choose one of the following:

- In the DynamoDB console, select your DAX cluster. The cluster endpoint is shown in the console, as in the following example.

  ```
  dax://my-cluster.l6fzcv.dax-clusters.us-east-1.amazonaws.com
  ```

- Using the AWS CLI, enter the following command:

  ```
  aws dax describe-clusters --query "Clusters[*].ClusterDiscoveryEndpoint"
  ```

  The cluster endpoint address, port, and URL are shown in the output, as in the following example.

  ```
  {
      "Address": "my-cluster.l6fzcv.dax-clusters.us-east-1.amazonaws.com",
      "Port": 8111,
      "URL": "dax://my-cluster.l6fzcv.dax-clusters.us-east-1.amazonaws.com"
  }
  ```

Now run the program again, but this time, specify the cluster endpoint URL as a command line parameter.

```
java -cp classpath TryDax dax://my-cluster.l6fzcv.dax-clusters.us-east-1.amazonaws.com
```

Look at the output and take note of the timing information. The elapsed times for `GetItem`and `Query` should be significantly lower with DAX than with DynamoDB.

SDK metrics

With DAX SDK for Java 2.x, you can collect metrics about the service clients in your application and analyze the output in Amazon CloudWatch. See Enabling SDK metrics for more information.

> **Note**
>
> The DAX SDK for Java only collects `ApiCallSuccessful` and `ApiCallDuration` metrics.

TryDax.java

```
import java.util.Map;

import software.amazon.awssdk.services.dynamodb.DynamoDbAsyncClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeDefinition;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.BillingMode;
import software.amazon.awssdk.services.dynamodb.model.CreateTableRequest;
import software.amazon.awssdk.services.dynamodb.model.DeleteTableRequest;
import software.amazon.awssdk.services.dynamodb.model.DescribeTableRequest;
import software.amazon.awssdk.services.dynamodb.model.GetItemRequest;
import software.amazon.awssdk.services.dynamodb.model.KeySchemaElement;
import software.amazon.awssdk.services.dynamodb.model.KeyType;
import software.amazon.awssdk.services.dynamodb.model.PutItemRequest;
import software.amazon.awssdk.services.dynamodb.model.QueryRequest;
import software.amazon.awssdk.services.dynamodb.model.ScalarAttributeType;
import software.amazon.dax.ClusterDaxAsyncClient;
import software.amazon.dax.Configuration;

public class TryDax {
    public static void main(String[] args) throws Exception {
        DynamoDbAsyncClient ddbClient = DynamoDbAsyncClient.builder()
                .build();

        DynamoDbAsyncClient daxClient = null;
        if (args.length >= 1) {
            daxClient = ClusterDaxAsyncClient.builder()
                    .overrideConfiguration(Configuration.builder()
```

```
                            .url(args[0]) // e.g. dax://my-cluster.l6fzcv.dax-
clusters.us-east-1.amazonaws.com
                            .build())
                    .build();
        }

        String tableName = "TryDaxTable";

        System.out.println("Creating table...");
        createTable(tableName, ddbClient);

        System.out.println("Populating table...");
        writeData(tableName, ddbClient, 100, 10);

        DynamoDbAsyncClient testClient = null;
        if (daxClient != null) {
            testClient = daxClient;
        } else {
            testClient = ddbClient;
        }

        System.out.println("Running GetItem and Query tests...");
        System.out.println("First iteration of each test will result in cache misses");
        System.out.println("Next iterations are cache hits\n");

        // GetItem
        getItemTest(tableName, testClient, 100, 10, 5);

        // Query
        queryTest(tableName, testClient, 100, 2, 9, 5);

        System.out.println("Deleting table...");
        deleteTable(tableName, ddbClient);
    }

    private static void createTable(String tableName, DynamoDbAsyncClient client) {
        try {
            System.out.println("Attempting to create table; please wait...");

            client.createTable(CreateTableRequest.builder()
                    .tableName(tableName)
                    .keySchema(KeySchemaElement.builder()
                            .keyType(KeyType.HASH)
                            .attributeName("pk")
```

```
                        .build(), KeySchemaElement.builder()
                        .keyType(KeyType.RANGE)
                        .attributeName("sk")
                        .build())
                .attributeDefinitions(AttributeDefinition.builder()
                        .attributeName("pk")
                        .attributeType(ScalarAttributeType.N)
                        .build(), AttributeDefinition.builder()
                        .attributeName("sk")
                        .attributeType(ScalarAttributeType.N)
                        .build())
                .billingMode(BillingMode.PAY_PER_REQUEST)
                .build()).get();
            client.waiter().waitUntilTableExists(DescribeTableRequest.builder()
                    .tableName(tableName)
                    .build()).get();
            System.out.println("Successfully created table.");

        } catch (Exception e) {
            System.err.println("Unable to create table: ");
            e.printStackTrace();
        }
    }

    private static void deleteTable(String tableName, DynamoDbAsyncClient client) {
        try {
            System.out.println("\nAttempting to delete table; please wait...");
            client.deleteTable(DeleteTableRequest.builder()
                    .tableName(tableName)
                    .build()).get();
            client.waiter().waitUntilTableNotExists(DescribeTableRequest.builder()
                    .tableName(tableName)
                    .build()).get();
            System.out.println("Successfully deleted table.");

        } catch (Exception e) {
            System.err.println("Unable to delete table: ");
            e.printStackTrace();
        }
    }

    private static void writeData(String tableName, DynamoDbAsyncClient client, int
 pkmax, int skmax) {
        System.out.println("Writing data to the table...");
```

```
        int stringSize = 1000;
        StringBuilder sb = new StringBuilder(stringSize);
        for (int i = 0; i < stringSize; i++) {
            sb.append('X');
        }
        String someData = sb.toString();

        try {
            for (int ipk = 1; ipk <= pkmax; ipk++) {
                System.out.println(("Writing " + skmax + " items for partition key: " +
ipk));
                for (int isk = 1; isk <= skmax; isk++) {
                    client.putItem(PutItemRequest.builder()
                            .tableName(tableName)
                            .item(Map.of("pk", attr(ipk), "sk", attr(isk), "someData",
attr(someData)))
                            .build()).get();
                }
            }
        } catch (Exception e) {
            System.err.println("Unable to write item:");
            e.printStackTrace();
        }
    }

    private static AttributeValue attr(int n) {
        return AttributeValue.builder().n(String.valueOf(n)).build();
    }

    private static AttributeValue attr(String s) {
        return AttributeValue.builder().s(s).build();
    }

    private static void getItemTest(String tableName, DynamoDbAsyncClient client, int
pk, int sk, int iterations) {
        long startTime, endTime;
        System.out.println("GetItem test - partition key 1-" + pk + " and sort keys 1-"
+ sk);

        for (int i = 0; i < iterations; i++) {
            startTime = System.nanoTime();
            try {
                for (int ipk = 1; ipk <= pk; ipk++) {
```

```
                    for (int isk = 1; isk <= sk; isk++) {
                        client.getItem(GetItemRequest.builder()
                                .tableName(tableName)
                                .key(Map.of("pk", attr(ipk), "sk", attr(isk)))
                                .build()).get();
                    }
                }
            } catch (Exception e) {
                System.err.println("Unable to get item:");
                e.printStackTrace();
            }
            endTime = System.nanoTime();
            printTime(startTime, endTime, pk * sk);
        }
    }

    private static void queryTest(String tableName, DynamoDbAsyncClient client, int pk,
int sk1, int sk2, int iterations) {
        long startTime, endTime;
        System.out.println("Query test - partition key 1-" + pk + " and sort keys
between " + sk1 + " and " + sk2);

        for (int i = 0; i < iterations; i++) {
            startTime = System.nanoTime();
            for (int ipk = 1; ipk <= pk; ipk++) {
                try {
                    // Pagination API for Query.
                    client.queryPaginator(QueryRequest.builder()
                            .tableName(tableName)
                            .keyConditionExpression("pk = :pkval and sk between :skval1
and :skval2")
                            .expressionAttributeValues(Map.of(":pkval", attr(ipk),
":skval1", attr(sk1), ":skval2", attr(sk2)))
                            .build()).items().subscribe((item) -> {
                    }).get();
                } catch (Exception e) {
                    System.err.println("Unable to query table:");
                    e.printStackTrace();
                }
            }
            endTime = System.nanoTime();
            printTime(startTime, endTime, pk);
        }
    }
```

```
    private static void printTime(long startTime, long endTime, int iterations) {
        System.out.format("\tTotal time: %.3f ms - ", (endTime - startTime) /
 (1000000.0));
        System.out.format("Avg time: %.3f ms\n", (endTime - startTime) / (iterations *
 1000000.0));
    }
}
```

.NET and DAX

Follow these steps to run the .NET sample on your Amazon EC2 instance.

> **Note**
>
> This tutorial uses the .NET 9 SDK. It shows how you can run a program in your default Amazon VPC to access your Amazon DynamoDB Accelerator (DAX) cluster. It works with the AWS SDK v4 for .NET . For details about changes in V4 and information about migrating, see Migrating to version 4 of the AWS SDK for .NET. If you prefer, you can use the AWS Toolkit for Visual Studio to write a .NET application and deploy it into your VPC. For more information, see Creating and Deploying Elastic Beanstalk Applications in .NET Using AWS Toolkit for Visual Studio in the *AWS Elastic Beanstalk Developer Guide*.

To run the .NET sample for DAX

1. Go to the Microsoft Downloads page and download the latest .NET 9 SDK for Linux. The downloaded file is `dotnet-sdk-N.N.N-linux-x64.tar.gz`.
2. Extract the SDK files.

   ```
   mkdir dotnet
   tar zxvf dotnet-sdk-N.N.N-linux-x64.tar.gz -C dotnet
   ```

   Replace *N.N.N* with the actual version number of the .NET SDK (for example: `9.0.305`).
3. Verify the installation.

   ```
   alias dotnet=$HOME/dotnet/dotnet
   dotnet --version
   ```

This should print the version number of the .NET SDK.

> **Note**
>
> Instead of the version number, you might receive the following error:
> error: libunwind.so.8: cannot open shared object file: No such file or directory
> To resolve the error, install the `libunwind` package.
>
> ```
> sudo yum install -y libunwind
> ```
>
> After you do this, you should be able to run the `dotnet --version` command without any errors.

4. Create a new .NET project.

```
dotnet new console -o myApp
```

This requires a few minutes to perform a one-time-only setup. When it is complete, run the sample project.

```
dotnet run --project myApp
```

You should receive the following message: `Hello World!`

5. The `myApp/myApp.csproj` file contains metadata about your project. To use the DAX client in your application, modify the file so that it looks like the following.

```
<Project Sdk="Microsoft.NET.Sdk">
    <PropertyGroup>
        <OutputType>Exe</OutputType>
        <TargetFramework>net9.0</TargetFramework>
    </PropertyGroup>
    <ItemGroup>
        <PackageReference Include="AWSSDK.DAX.Client" Version="*" />
    </ItemGroup>
</Project>
```

6. Download the sample program source code (`.zip` file).

```
wget http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/samples/
TryDax.zip
```

When the download is complete, extract the source files.

```
unzip TryDax.zip
```

7. Now run the sample programs of *dotNet*, one at a time. For each program, copy its contents into the `myApp/Program.cs`, and then run the `MyApp` project.

   Run the following .NET programs. The first program creates a DynamoDB table named `TryDaxTable`. The second program writes data to the table.

```
cp TryDax/dotNet/01-CreateTable.cs myApp/Program.cs
dotnet run --project myApp

cp TryDax/dotNet/02-Write-Data.cs myApp/Program.cs
dotnet run --project myApp
```

8. Next, run some programs to perform `GetItem`, `Query`, and `Scan` operations on your DAX cluster. To determine the endpoint for your DAX cluster, choose one of the following:

   - **Using the DynamoDB console** — Choose your DAX cluster. The cluster endpoint is shown on the console, as in the following example.

```
dax://my-cluster.l6fzcv.dax-clusters.us-east-1.amazonaws.com
```

   - **Using the AWS CLI** — Enter the following command.

```
aws dax describe-clusters --query "Clusters[*].ClusterDiscoveryEndpoint"
```

     The cluster endpoint is shown in the output, as in the following example.

```
{
    "Address": "my-cluster.l6fzcv.dax-clusters.us-east-1.amazonaws.com",
    "Port": 8111,
    "URL": "dax://my-cluster.l6fzcv.dax-clusters.us-east-1.amazonaws.com"
}
```

Now run the following programs, specifying your cluster endpoint as a command line parameter. (Replace the sample endpoint with your actual DAX cluster endpoint.)

```
cp TryDax/dotNet/03-GetItem-Test.cs myApp/Program.cs
dotnet run --project myApp dax://my-cluster.l6fzcv.dax-clusters.us-
east-1.amazonaws.com

cp TryDax/dotNet/04-Query-Test.cs myApp/Program.cs
dotnet run --project myApp dax://my-cluster.l6fzcv.dax-clusters.us-
east-1.amazonaws.com

cp TryDax/dotNet/05-Scan-Test.cs myApp/Program.cs
dotnet run --project myApp dax://my-cluster.l6fzcv.dax-clusters.us-
east-1.amazonaws.com
```

Take note of the timing information—the number of milliseconds required for the `GetItem`, `Query`, and `Scan` tests.

9. Run the following .NET program to delete `TryDaxTable`.

```
cp TryDax/dotNet/06-DeleteTable.cs myApp/Program.cs
dotnet run --project myApp
```

For more information about these programs, see the following sections:

- 01-CreateTable.cs
- 02-Write-Data.cs
- 03-GetItem-Test.cs
- 04-Query-Test.cs
- 05-Scan-Test.cs
- 06-DeleteTable.cs

01-CreateTable.cs

The `01-CreateTable.cs` program creates a table (`TryDaxTable`). The remaining .NET programs in this section depend on this table.

```
using System;
using System.Collections.Generic;
using System.Threading.Tasks;
using Amazon.DynamoDBv2;
using Amazon.DynamoDBv2.Model;

namespace ClientTest
{
    class Program
    {
        public static async Task Main(string[] args)
        {
            AmazonDynamoDBClient client = new AmazonDynamoDBClient();

            var tableName = "TryDaxTable";

            var request = new CreateTableRequest()
            {
                TableName = tableName,
                KeySchema = new List<KeySchemaElement>()
                {
                    new KeySchemaElement{ AttributeName = "pk",KeyType = "HASH"},
                    new KeySchemaElement{ AttributeName = "sk",KeyType = "RANGE"}
                },
                AttributeDefinitions = new List<AttributeDefinition>() {
                    new AttributeDefinition{ AttributeName = "pk",AttributeType = "N"},
                    new AttributeDefinition{ AttributeName = "sk",AttributeType  = "N"}
                },
                ProvisionedThroughput = new ProvisionedThroughput()
                {
                    ReadCapacityUnits = 10,
                    WriteCapacityUnits = 10
                }
            };

            var response = await client.CreateTableAsync(request);

            Console.WriteLine("Hit <enter> to continue...");
            Console.ReadLine();
        }
    }
}
```

02-Write-Data.cs

The `02-Write-Data.cs` program writes test data to `TryDaxTable`.

```
using System;
using System.Collections.Generic;
using System.Threading.Tasks;
using Amazon.DynamoDBv2;
using Amazon.DynamoDBv2.Model;

namespace ClientTest
{
    class Program
    {
        public static async Task Main(string[] args)
        {
            AmazonDynamoDBClient client = new AmazonDynamoDBClient();

            var tableName = "TryDaxTable";

            string someData = new string('X', 1000);
            var pkmax = 10;
            var skmax = 10;

            for (var ipk = 1; ipk <= pkmax; ipk++)
            {
                Console.WriteLine($"Writing {skmax} items for partition key: {ipk}");
                for (var isk = 1; isk <= skmax; isk++)
                {
                    var request = new PutItemRequest()
                    {
                        TableName = tableName,
                        Item = new Dictionary<string, AttributeValue>()
                       {
                            { "pk", new AttributeValue{N = ipk.ToString() } },
                            { "sk", new AttributeValue{N = isk.ToString() } },
                            { "someData", new AttributeValue{S = someData } }
                       }
                    };

                    var response = await client.PutItemAsync(request);
```

```
                }
            }

            Console.WriteLine("Hit <enter> to continue...");
            Console.ReadLine();
        }
    }
}
```

03-GetItem-Test.cs

The `03-GetItem-Test.cs` program performs `GetItem` operations on `TryDaxTable`.

```
using System;
using System.Collections.Generic;
using System.Threading.Tasks;
using Amazon.DAX;
using Amazon.DynamoDBv2.Model;
using Amazon.Runtime;

namespace ClientTest
{
    class Program
    {
        public static async Task Main(string[] args)
        {
            string endpointUri = args[0];
            Console.WriteLine($"Using DAX client - endpointUri={endpointUri}");

            var clientConfig = new DaxClientConfig(endpointUri)
            {
                AwsCredentials = FallbackCredentialsFactory.GetCredentials()
            };
            var client = new ClusterDaxClient(clientConfig);

            var tableName = "TryDaxTable";

            var pk = 1;
            var sk = 10;
            var iterations = 5;

            var startTime = System.DateTime.Now;
```

```
            for (var i = 0; i < iterations; i++)
            {
                for (var ipk = 1; ipk <= pk; ipk++)
                {
                    for (var isk = 1; isk <= sk; isk++)
                    {
                        var request = new GetItemRequest()
                        {
                            TableName = tableName,
                            Key = new Dictionary<string, AttributeValue>() {
                            {"pk", new AttributeValue {N = ipk.ToString()} },
                            {"sk", new AttributeValue {N = isk.ToString() } }
                        }
                        };
                        var response = await client.GetItemAsync(request);
                        Console.WriteLine($"GetItem succeeded for pk: {ipk},sk:
 {isk}");
                    }
                }
            }

            var endTime = DateTime.Now;
            TimeSpan timeSpan = endTime - startTime;
            Console.WriteLine($"Total time: {timeSpan.TotalMilliseconds}
 milliseconds");

            Console.WriteLine("Hit <enter> to continue...");
            Console.ReadLine();
        }
    }
}
```

04-Query-Test.cs

The `04-Query-Test.cs` program performs `Query` operations on `TryDaxTable`.

```
using System;
using System.Collections.Generic;
using System.Threading.Tasks;
using Amazon.Runtime;
using Amazon.DAX;
```

```
using Amazon.DynamoDBv2.Model;

namespace ClientTest
{
    class Program
    {
        public static async Task Main(string[] args)
        {
            string endpointUri = args[0];
            Console.WriteLine($"Using DAX client - endpointUri={endpointUri}");


            var clientConfig = new DaxClientConfig(endpointUri)
            {
                AwsCredentials = FallbackCredentialsFactory.GetCredentials()
            };
            var client = new ClusterDaxClient(clientConfig);

            var tableName = "TryDaxTable";

            var pk = 5;
            var sk1 = 2;
            var sk2 = 9;
            var iterations = 5;

            var startTime = DateTime.Now;

            for (var i = 0; i < iterations; i++)
            {
                var request = new QueryRequest()
                {
                    TableName = tableName,
                    KeyConditionExpression = "pk = :pkval and sk between :skval1
 and :skval2",
                    ExpressionAttributeValues = new Dictionary<string,
 AttributeValue>() {
                            {":pkval", new AttributeValue {N = pk.ToString()} },
                            {":skval1", new AttributeValue {N = sk1.ToString()} },
                            {":skval2", new AttributeValue {N = sk2.ToString()} }
                    }
                };
                var response = await client.QueryAsync(request);
                Console.WriteLine($"{i}: Query succeeded");
```

```
            }

            var endTime = DateTime.Now;
            TimeSpan timeSpan = endTime - startTime;
            Console.WriteLine($"Total time: {timeSpan.TotalMilliseconds}
 milliseconds");

            Console.WriteLine("Hit <enter> to continue...");
            Console.ReadLine();
        }
    }
}
```

05-Scan-Test.cs

The `05-Scan-Test.cs` program performs `Scan` operations on `TryDaxTable`.

```
using System;
using System.Threading.Tasks;
using Amazon.Runtime;
using Amazon.DAX;
using Amazon.DynamoDBv2.Model;

namespace ClientTest
{
    class Program
    {
        public static async Task Main(string[] args)
        {
            string endpointUri = args[0];
            Console.WriteLine($"Using DAX client - endpointUri={endpointUri}");

            var clientConfig = new DaxClientConfig(endpointUri)
            {
                AwsCredentials = FallbackCredentialsFactory.GetCredentials()
            };
            var client = new ClusterDaxClient(clientConfig);

            var tableName = "TryDaxTable";

            var iterations = 5;
```

```
            var startTime = DateTime.Now;

            for (var i = 0; i < iterations; i++)
            {
                var request = new ScanRequest()
                {
                    TableName = tableName
                };
                var response = await client.ScanAsync(request);
                Console.WriteLine($"{i}: Scan succeeded");
            }

            var endTime = DateTime.Now;
            TimeSpan timeSpan = endTime - startTime;
            Console.WriteLine($"Total time: {timeSpan.TotalMilliseconds}
 milliseconds");

            Console.WriteLine("Hit <enter> to continue...");
            Console.ReadLine();
        }
    }
}
```

06-DeleteTable.cs

The `06-DeleteTable.cs` program deletes `TryDaxTable`. Run this program after you have finished testing.

```
using System;
using System.Threading.Tasks;
using Amazon.DynamoDBv2.Model;
using Amazon.DynamoDBv2;

namespace ClientTest
{
    class Program
    {
        public static async Task Main(string[] args)
        {
            AmazonDynamoDBClient client = new AmazonDynamoDBClient();
```

```
            var tableName = "TryDaxTable";

            var request = new DeleteTableRequest()
            {
                TableName = tableName
            };

            var response = await client.DeleteTableAsync(request);

            Console.WriteLine("Hit <enter> to continue...");
            Console.ReadLine();
        }
    }
}
```

Python and DAX

Follow this procedure to run the Python sample application on your Amazon EC2 instance.

To run the Python sample for DAX

1. Install the DAX Python client using the `pip` utility.

   ```
   pip install amazon-dax-client
   ```

2. Download the sample program source code (`.zip` file).

   ```
   wget http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/samples/
   TryDax.zip
   ```

   When the download is complete, extract the source files.

   ```
   unzip TryDax.zip
   ```

3. Run the following Python programs. The first program creates an Amazon DynamoDB table named `TryDaxTable`. The second program writes data to the table.

   ```
   python 01-create-table.py
   python 02-write-data.py
   ```

4. Run the following Python programs.

```
python 03-getitem-test.py
python 04-query-test.py
python 05-scan-test.py
```

Take note of the timing information—the number of milliseconds required for the `GetItem`, `Query`, and `Scan` tests.

5. In the previous step, you ran the programs against the DynamoDB endpoint. Now run the programs again, but this time, the `GetItem`, `Query`, and `Scan` operations are processed by your DAX cluster.

   To determine the endpoint for your DAX cluster, choose one of the following:

   - **Using the DynamoDB console** — Choose your DAX cluster. The cluster endpoint is shown on the console, as in the following example.

     ```
     dax://my-cluster.l6fzcv.dax-clusters.us-east-1.amazonaws.com
     ```

   - **Using the AWS CLI** — Enter the following command.

     ```
     aws dax describe-clusters --query "Clusters[*].ClusterDiscoveryEndpoint"
     ```

     The cluster endpoint is shown in the output, as in this example.

     ```
     {
         "Address": "my-cluster.l6fzcv.dax-clusters.us-east-1.amazonaws.com",
         "Port": 8111,
         "URL": "dax://my-cluster.l6fzcv.dax-clusters.us-east-1.amazonaws.com"
     }
     ```

   Run the programs again, but this time, specify the cluster endpoint as a command line parameter.

   ```
   python 03-getitem-test.py dax://my-cluster.l6fzcv.dax-clusters.us-
   east-1.amazonaws.com
   python 04-query-test.py dax://my-cluster.l6fzcv.dax-clusters.us-
   east-1.amazonaws.com
   python 05-scan-test.py dax://my-cluster.l6fzcv.dax-clusters.us-east-1.amazonaws.com
   ```

Look at the rest of the output, and take note of the timing information. The elapsed times for `GetItem`, `Query`, and `Scan` should be significantly lower with DAX than with DynamoDB.

6. Run the following Python program to delete `TryDaxTable`.

```
python 06-delete-table.py
```

For more information about these programs, see the following sections:

- 01-create-table.py
- 02-write-data.py
- 03-getitem-test.py
- 04-query-test.py
- 05-scan-test.py
- 06-delete-table.py

01-create-table.py

The `01-create-table.py` program creates a table (`TryDaxTable`). The remaining Python programs in this section depend on this table.

```
import boto3


def create_dax_table(dyn_resource=None):
    """
    Creates a DynamoDB table.

    :param dyn_resource: Either a Boto3 or DAX resource.
    :return: The newly created table.
    """
    if dyn_resource is None:
        dyn_resource = boto3.resource("dynamodb")

    table_name = "TryDaxTable"
    params = {
        "TableName": table_name,
        "KeySchema": [
```

```
            {"AttributeName": "partition_key", "KeyType": "HASH"},
            {"AttributeName": "sort_key", "KeyType": "RANGE"},
        ],
        "AttributeDefinitions": [
            {"AttributeName": "partition_key", "AttributeType": "N"},
            {"AttributeName": "sort_key", "AttributeType": "N"},
        ],
        "BillingMode": "PAY_PER_REQUEST",
    }
    table = dyn_resource.create_table(**params)
    print(f"Creating {table_name}...")
    table.wait_until_exists()
    return table


if __name__ == "__main__":
    dax_table = create_dax_table()
    print(f"Created table.")
```

02-write-data.py

The `02-write-data.py` program writes test data to `TryDaxTable`.

```
import boto3


def write_data_to_dax_table(key_count, item_size, dyn_resource=None):
    """
    Writes test data to the demonstration table.

    :param key_count: The number of partition and sort keys to use to populate the
                      table. The total number of items is key_count * key_count.
    :param item_size: The size of non-key data for each test item.
    :param dyn_resource: Either a Boto3 or DAX resource.
    """
    if dyn_resource is None:
        dyn_resource = boto3.resource("dynamodb")

    table = dyn_resource.Table("TryDaxTable")
    some_data = "X" * item_size

    for partition_key in range(1, key_count + 1):
        for sort_key in range(1, key_count + 1):
            table.put_item(
```

```
                Item={
                    "partition_key": partition_key,
                    "sort_key": sort_key,
                    "some_data": some_data,
                }
            )
            print(f"Put item ({partition_key}, {sort_key}) succeeded.")


if __name__ == "__main__":
    write_key_count = 10
    write_item_size = 1000
    print(
        f"Writing {write_key_count*write_key_count} items to the table. "
        f"Each item is {write_item_size} characters."
    )
    write_data_to_dax_table(write_key_count, write_item_size)
```

03-getitem-test.py

The `03-getitem-test.py` program performs `GetItem` operations on `TryDaxTable`. This example is given for the Region eu-west-1.

```
import argparse
import sys
import time
import amazondax
import boto3


def get_item_test(key_count, iterations, dyn_resource=None):
    """
    Gets items from the table a specified number of times. The time before the
    first iteration and the time after the last iteration are both captured
    and reported.

    :param key_count: The number of items to get from the table in each iteration.
    :param iterations: The number of iterations to run.
    :param dyn_resource: Either a Boto3 or DAX resource.
    :return: The start and end times of the test.
    """
    if dyn_resource is None:
```

```
        dyn_resource = boto3.resource('dynamodb')

    table = dyn_resource.Table('TryDaxTable')
    start = time.perf_counter()
    for _ in range(iterations):
        for partition_key in range(1, key_count + 1):
            for sort_key in range(1, key_count + 1):
                table.get_item(Key={
                    'partition_key': partition_key,
                    'sort_key': sort_key
                })
                print('.', end='')
                sys.stdout.flush()
    print()
    end = time.perf_counter()
    return start, end


if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument(
        'endpoint_url', nargs='?',
        help="When specified, the DAX cluster endpoint. Otherwise, DAX is not used.")
    args = parser.parse_args()

    test_key_count = 10
    test_iterations = 50
    if args.endpoint_url:
        print(f"Getting each item from the table {test_iterations} times, "
              f"using the DAX client.")
        # Use a with statement so the DAX client closes the cluster after completion.
        with amazondax.AmazonDaxClient.resource(endpoint_url=args.endpoint_url,
 region_name='eu-west-1') as dax:
            test_start, test_end = get_item_test(
                test_key_count, test_iterations, dyn_resource=dax)
    else:
        print(f"Getting each item from the table {test_iterations} times, "
              f"using the Boto3 client.")
        test_start, test_end = get_item_test(
            test_key_count, test_iterations)
    print(f"Total time: {test_end - test_start:.4f} sec. Average time: "
          f"{(test_end - test_start)/ test_iterations}.")
```

04-query-test.py

The `04-query-test.py` program performs `Query` operations on `TryDaxTable`.

```
import argparse
import time
import sys
import amazondax
import boto3
from boto3.dynamodb.conditions import Key


def query_test(partition_key, sort_keys, iterations, dyn_resource=None):
    """
    Queries the table a specified number of times. The time before the
    first iteration and the time after the last iteration are both captured
    and reported.

    :param partition_key: The partition key value to use in the query. The query
                          returns items that have partition keys equal to this value.
    :param sort_keys: The range of sort key values for the query. The query returns
                      items that have sort key values between these two values.
    :param iterations: The number of iterations to run.
    :param dyn_resource: Either a Boto3 or DAX resource.
    :return: The start and end times of the test.
    """
    if dyn_resource is None:
        dyn_resource = boto3.resource("dynamodb")

    table = dyn_resource.Table("TryDaxTable")
    key_condition_expression = Key("partition_key").eq(partition_key) & Key(
        "sort_key"
    ).between(*sort_keys)

    start = time.perf_counter()
    for _ in range(iterations):
        table.query(KeyConditionExpression=key_condition_expression)
        print(".", end="")
        sys.stdout.flush()
    print()
    end = time.perf_counter()
    return start, end
```

```
if __name__ == "__main__":
    # pylint: disable=not-context-manager
    parser = argparse.ArgumentParser()
    parser.add_argument(
        "endpoint_url",
        nargs="?",
        help="When specified, the DAX cluster endpoint. Otherwise, DAX is not used.",
    )
    args = parser.parse_args()

    test_partition_key = 5
    test_sort_keys = (2, 9)
    test_iterations = 100
    if args.endpoint_url:
        print(f"Querying the table {test_iterations} times, using the DAX client.")
        # Use a with statement so the DAX client closes the cluster after completion.
        with amazondax.AmazonDaxClient.resource(endpoint_url=args.endpoint_url) as dax:
            test_start, test_end = query_test(
                test_partition_key, test_sort_keys, test_iterations, dyn_resource=dax
            )
    else:
        print(f"Querying the table {test_iterations} times, using the Boto3 client.")
        test_start, test_end = query_test(
            test_partition_key, test_sort_keys, test_iterations
        )

    print(
        f"Total time: {test_end - test_start:.4f} sec. Average time: "
        f"{(test_end - test_start)/test_iterations}."
    )
```

05-scan-test.py

The `05-scan-test.py` program performs `Scan` operations on `TryDaxTable`.

```
import argparse
import time
import sys
import amazondax
import boto3


def scan_test(iterations, dyn_resource=None):
    """
```

```
    Scans the table a specified number of times. The time before the
    first iteration and the time after the last iteration are both captured
    and reported.

    :param iterations: The number of iterations to run.
    :param dyn_resource: Either a Boto3 or DAX resource.
    :return: The start and end times of the test.
    """
    if dyn_resource is None:
        dyn_resource = boto3.resource("dynamodb")

    table = dyn_resource.Table("TryDaxTable")
    start = time.perf_counter()
    for _ in range(iterations):
        table.scan()
        print(".", end="")
        sys.stdout.flush()
    print()
    end = time.perf_counter()
    return start, end


if __name__ == "__main__":
    # pylint: disable=not-context-manager
    parser = argparse.ArgumentParser()
    parser.add_argument(
        "endpoint_url",
        nargs="?",
        help="When specified, the DAX cluster endpoint. Otherwise, DAX is not used.",
    )
    args = parser.parse_args()

    test_iterations = 100
    if args.endpoint_url:
        print(f"Scanning the table {test_iterations} times, using the DAX client.")
        # Use a with statement so the DAX client closes the cluster after completion.
        with amazondax.AmazonDaxClient.resource(endpoint_url=args.endpoint_url) as dax:
            test_start, test_end = scan_test(test_iterations, dyn_resource=dax)
    else:
        print(f"Scanning the table {test_iterations} times, using the Boto3 client.")
        test_start, test_end = scan_test(test_iterations)
    print(
        f"Total time: {test_end - test_start:.4f} sec. Average time: "
        f"{(test_end - test_start)/test_iterations}."
```

```
    )
```

06-delete-table.py

The `06-delete-table.py` program deletes `TryDaxTable`. Run this program after you have finished testing Amazon DynamoDB Accelerator (DAX) functionality.

```
import boto3


def delete_dax_table(dyn_resource=None):
    """
    Deletes the demonstration table.

    :param dyn_resource: Either a Boto3 or DAX resource.
    """
    if dyn_resource is None:
        dyn_resource = boto3.resource("dynamodb")

    table = dyn_resource.Table("TryDaxTable")
    table.delete()

    print(f"Deleting {table.name}...")
    table.wait_until_not_exists()


if __name__ == "__main__":
    delete_dax_table()
    print("Table deleted!")
```

Modifying an existing application to use DAX

If you already have a Java application that uses Amazon DynamoDB, you can modify it so that it can access your DynamoDB Accelerator (DAX) cluster. You don't have to rewrite the entire application because the DAX Java client is similar to the DynamoDB low-level client included in the AWS SDK for Java 2.x. See Working with items in DynamoDB for details.

> **Note**
>
> This example uses AWS SDK for Java 2.x. For the legacy SDK for Java 1.x version, see Modifying an existing SDK for Java 1.x application to use DAX.

To modify your program, replace the DynamoDB client with a DAX client.

```
Region region = Region.US_EAST_1;

// Create an asynchronous DynamoDB client
DynamoDbAsyncClient client = DynamoDbAsyncClient.builder()
                .region(region)
                .build();

// Create an asynchronous DAX client
DynamoDbAsyncClient client = ClusterDaxAsyncClient.builder()
                .overrideConfiguration(Configuration.builder()
                    .url(<cluster url>) // for example, "dax://my-cluster.l6fzcv.dax-
clusters.us-east-1.amazonaws.com"
                    .region(region)
                    .addMetricPublisher(cloudWatchMetricsPub) // optionally enable SDK
 metric collection
                    .build())
                .build();
```

You can also use the high-level library that is part of the AWS SDK for Java 2.x, replacing the DynamoDB client with a DAX client.

```
Region region = Region.US_EAST_1;
DynamoDbAsyncClient dax = ClusterDaxAsyncClient.builder()
        .overrideConfiguration(Configuration.builder()
            .url(<cluster url>) // for example, "dax://my-cluster.l6fzcv.dax-
clusters.us-east-1.amazonaws.com"
            .region(region)
            .build())
        .build();

DynamoDbEnhancedAsyncClient enhancedClient = DynamoDbEnhancedAsyncClient.builder()
        .dynamoDbClient(dax)
        .build();
```

For more information, see Mapping items in DynamoDB tables.

Managing DAX clusters

This section addresses some of the common management tasks for Amazon DynamoDB Accelerator (DAX) clusters.

Topics

- IAM permissions for managing a DAX cluster
- Scaling a DAX cluster
- Customizing DAX cluster settings
- Configuring TTL settings
- Tagging support for DAX
- AWS CloudTrail integration
- Deleting a DAX cluster

IAM permissions for managing a DAX cluster

When you administer a DAX cluster using the AWS Management Console or the AWS Command Line Interface (AWS CLI), we strongly recommend that you narrow the scope of actions that users can perform. By doing so, you help mitigate risk while following the principle of least privilege.

The following discussion focuses on access control for the DAX management APIs. For more information, see Amazon DynamoDB accelerator in the *Amazon DynamoDB API Reference*.

> **Note**
>
> For more detailed information about managing AWS Identity and Access Management (IAM) permissions, see the following:
>
> - IAM and creating DAX clusters: Creating a DAX cluster.
> - IAM and DAX data plane operations: DAX access control.

For the DAX management APIs, you can't scope API actions to a specific resource. The `Resource` element must be set to `"*"`. This is different from DAX data plane API operations, such as `GetItem`, `Query`, and `Scan`. Data plane operations are exposed through the DAX client, and those operations *can* be scoped to specific resources.

To illustrate, consider the following IAM policy document.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Action": [
                "dax:*"
            ],
            "Effect": "Allow",
            "Resource": [
                "arn:aws:dax:us-west-2:123456789012:cache/DAXCluster01"
            ]
        }
    ]
}
```

Suppose that the intent of this policy is to allow DAX management API calls for the cluster `DAXCluster01`— and only that cluster.

Now suppose that a user issues the following AWS CLI command.

```
aws dax describe-clusters
```

This command fails with a Not Authorized exception because the underlying `DescribeClusters` API call can't be scoped to a specific cluster. Even though the policy is syntactically valid, the command fails because the `Resource` element must be set to `"*"`. However, if the user runs a program that sends DAX data plane calls (such as `GetItem` or `Query`) to `DAXCluster01`, those calls *do* succeed. This is because DAX data plane APIs can be scoped to specific resources (in this case, `DAXCluster01`).

If you want to write a single comprehensive IAM policy to encompass both DAX management APIs and DAX data plane APIs, we suggest that you include two distinct statements in the policy document. One of these statements should address the DAX data plane APIs, while the other statement addresses the DAX management APIs.

The following example policy shows this approach. Note how the `DAXDataAPIs` statement is scoped to the `DAXCluster01` resource, but the resource for `DAXManagementAPIs` must be `"*"`.

The actions shown in each statement are for illustration only. You can customize them as needed for your application.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "DAXDataAPIs",
            "Action": [
                "dax:GetItem",
                "dax:BatchGetItem",
                "dax:Query",
                "dax:Scan",
                "dax:PutItem",
                "dax:UpdateItem",
                "dax:DeleteItem",
                "dax:BatchWriteItem"
            ],
            "Effect": "Allow",
            "Resource": [
                "arn:aws:dax:us-west-2:123456789012:cache/DAXCluster01"
            ]},
            {
            "Sid": "DAXManagementAPIs",
            "Action": [
                "dax:CreateParameterGroup",
                "dax:CreateSubnetGroup",
                "dax:DecreaseReplicationFactor",
                "dax:DeleteCluster",
                "dax:DeleteParameterGroup",
                "dax:DeleteSubnetGroup",
                "dax:DescribeClusters",
                "dax:DescribeDefaultParameters",
                "dax:DescribeEvents",
                "dax:DescribeParameterGroups",
                "dax:DescribeParameters",
                "dax:DescribeSubnetGroups",
                "dax:IncreaseReplicationFactor",
                "dax:ListTags",
                "dax:RebootNode",
                "dax:TagResource",
```

```
                "dax:UntagResource",
                "dax:UpdateCluster",
                "dax:UpdateParameterGroup",
                "dax:UpdateSubnetGroup"
                ],
            "Effect": "Allow",
            "Resource": [
                "*"
            ]
        }
    ]
}
```

Scaling a DAX cluster

There are two options available for scaling a DAX cluster. The first option is *horizontal scaling*, where you add read replicas to the cluster. The second option is *vertical scaling*, where you select different node types. For advice on how to approach choosing an appropriate cluster size and node type for your application, see DAX cluster sizing guide.

Horizontal scaling

With horizontal scaling, you can improve throughput for read operations by adding more read replicas to the cluster. A single DAX cluster supports up to 10 read replicas, and you can add or remove replicas while the cluster is running.

When you add a new node, you must synchronize the cache data from a peer node. Therefore, the addition time varies based on cache size and your application workload. As a best practice, we recommend that you pre-scale your cluster to meet expected traffic peaks. For information about right-sizing guidelines and monitoring recommendations, see DAX cluster sizing guide.

The following AWS CLI examples show how to increase or decrease the number of nodes. The `--new-replication-factor` argument specifies the total number of nodes in the cluster. One of the nodes is the primary node, and the other nodes are read replicas.

```
aws dax increase-replication-factor \
    --cluster-name MyNewCluster  \
    --new-replication-factor 5
```

```
aws dax decrease-replication-factor \
```

```
    --cluster-name MyNewCluster  \
    --new-replication-factor 3
```

> **Note**
>
> The cluster status changes to `modifying` when you modify the replication factor. The status changes to `available` when the modification is complete.

Vertical scaling

If you have a large working set of data, your application might benefit from using larger node types. Larger nodes can enable the cluster to store more data in memory, reducing cache misses and improving overall application performance of the application. (All of the nodes in a DAX cluster must be of the same type.)

If your DAX cluster has a high rate of write operations or cache misses, your application might also benefit from using larger node types. Write operations and cache misses consume resources on the cluster's primary node. Therefore, using larger node types might increase the performance of the primary node and thereby allow a higher throughput for these types of operations.

You can't modify the node types on a running DAX cluster. Instead, you must create a new cluster with the desired node type. For a list of supported node types, see Nodes.

You can create a new DAX cluster using the AWS Management Console, CloudFormation, the AWS CLI, or the AWS SDK. (For the AWS CLI, use the `--node-type` parameter to specify the node type.)

Customizing DAX cluster settings

When you create a DAX cluster, the following default settings are used:

- Automatic cache eviction enabled with Time to Live (TTL) of 5 minutes
- No preference for Availability Zones
- No preference for maintenance windows
- Notifications disabled

For new clusters, you can customize the settings at creation time. To do this in the AWS Management Console, clear **Use default settings** to modify the following settings:

- **Network and Security**—Allows you to run individual DAX cluster nodes in different Availability Zones within the current AWS Region. If you choose **No Preference**, the nodes are distributed among Availability Zones automatically.
- **Parameter Group**—A named set of parameters that are applied to every node in the cluster. You can use a parameter group to specify cache TTL behavior. You can change the value of any given parameter within a parameter group (except default parameter group `default.dax.1.0`) at any time.
- **Maintenance Window**—A weekly time period during which software upgrades and patches are applied to the nodes in the cluster. You can choose the start day, start time, and duration of the maintenance window. If you choose **No Preference**, the maintenance window is selected at random from an 8-hour block of time per Region. For more information, see Maintenance window.

> **Note**
>
> **Parameter Group** and **Maintenance Window** can also be changed at any time on a running cluster.

When a maintenance event occurs, DAX can notify you using Amazon Simple Notification Service (Amazon SNS). To configure notifications, choose an option from the **Topic for SNS notification** selector. You can create a new Amazon SNS topic, or use an existing topic.

For more information about setting up and subscribing to an Amazon SNS topic, see Getting started with Amazon SNS in the *Amazon Simple Notification Service Developer Guide*.

Configuring TTL settings

DAX maintains two caches for data that it reads from DynamoDB:

- **Item cache**—For items retrieved using `GetItem` or `BatchGetItem`.
- **Query cache**—For result sets retrieved using `Query` or `Scan`.

For more information, see Item cache and Query cache.

The default TTL for each of these caches is 5 minutes. If you want to use different TTL settings, you can launch a DAX cluster using a custom parameter group. To do this on the console, choose **DAX | Parameter groups** in the navigation pane.

You can also perform these tasks using the AWS CLI. The following example shows how to launch a new DAX cluster using a custom parameter group. In this example, the item cache TTL is set to 10 minutes, and the query cache TTL is set to 3 minutes.

1. Create a new parameter group.

```
aws dax create-parameter-group \
    --parameter-group-name custom-ttl
```

2. Set the item cache TTL to 10 minutes (600000 milliseconds).

```
aws dax update-parameter-group \
    --parameter-group-name custom-ttl \
    --parameter-name-values "ParameterName=record-ttl-millis,ParameterValue=600000"
```

3. Set the query cache TTL to 3 minutes (180000 milliseconds).

```
aws dax update-parameter-group \
    --parameter-group-name custom-ttl \
    --parameter-name-values "ParameterName=query-ttl-millis,ParameterValue=180000"
```

4. Verify that the parameters have been set correctly.

```
aws dax describe-parameters --parameter-group-name custom-ttl \
    --query "Parameters[*].[ParameterName,Description,ParameterValue]"
```

You can now launch a new DAX cluster with this parameter group.

```
aws dax create-cluster \
    --cluster-name MyNewCluster \
    --node-type dax.r3.large \
    --replication-factor 3 \
    --iam-role-arn arn:aws:iam::123456789012:role/DAXServiceRole \
    --parameter-group custom-ttl
```

> **Note**
>
> You can modify a parameter group attached to a running DAX cluster. Parameter changes apply only to items written after the modification. Existing cached items retain the TTL and settings from when they were originally written.

Tagging support for DAX

Many AWS services, including DynamoDB, support *tagging*—the ability to label resources with user-defined names. You can assign tags to DAX clusters, allowing you to quickly identify all of your AWS resources that have the same tag, or to categorize your AWS bills by the tags you assign.

For more information, see Adding tags and labels to resources in DynamoDB.

Using the AWS Management Console

To manage DAX cluster tags

1. Open the DynamoDB console at https://console.aws.amazon.com/dynamodb/.
2. In the navigation pane, under **DAX**, choose **Clusters**.
3. Choose the cluster that you want to work with.
4. Choose the **Tags** tab. You can add, list, edit, or delete your tags here.

   When the settings are as you want them, choose **Apply Changes**.

Using the AWS CLI

When you use the AWS CLI to manage DAX cluster tags, you must first determine the Amazon Resource Name (ARN) for the cluster. The following example shows how to determine the ARN for a cluster named `MyDAXCluster`.

```
aws dax describe-clusters \
    --cluster-name MyDAXCluster \
    --query "Clusters[*].ClusterArn"
```

In the output, the ARN will look similar to this: `arn:aws:dax:us-west-2:123456789012:cache/MyDAXCluster`

The following example shows how to tag the cluster.

```
aws dax tag-resource  \
    --resource-name arn:aws:dax:us-west-2:123456789012:cache/MyDAXCluster \
    --tags="Key=ClusterUsage,Value=prod"
```

List all the tags for a cluster.

```
aws dax list-tags \
    --resource-name arn:aws:dax:us-west-2:123456789012:cache/MyDAXCluster
```

To remove a tag, specify its key.

```
aws dax untag-resource  \
    --resource-name arn:aws:dax:us-west-2:123456789012:cache/MyDAXCluster \
    --tag-keys ClusterUsage
```

AWS CloudTrail integration

DAX is integrated with AWS CloudTrail, allowing you to audit DAX cluster activities. You can use CloudTrail logs to view all the changes that have been made at the cluster level. You can also see changes to cluster components such as nodes, subnet groups, and parameter groups. For more information, see Logging DynamoDB operations by using AWS CloudTrail.

Deleting a DAX cluster

If you are no longer using a DAX cluster, you should delete it to avoid being charged for unused resources.

You can delete a DAX cluster using the console or the AWS CLI. The following is an example.

```
aws dax delete-cluster --cluster-name mydaxcluster
```

Monitoring DynamoDB Accelerator

Monitoring is an important part of maintaining the reliability, availability, and performance of Amazon DynamoDB Accelerator (DAX) and your AWS solutions. You should collect monitoring data

from all parts of your AWS solution so that you can more easily debug a multi-point failure, if one occurs.

Before you start monitoring DAX, you should create a monitoring plan that includes answers to the following questions:

- What are your monitoring goals?
- What resources will you monitor?
- How often will you monitor these resources?
- What monitoring tools will you use?
- Who will perform the monitoring tasks?
- Who should be notified when something goes wrong?

Topics

- Monitoring tools for DynamoDB Accelerator
- Monitoring with Amazon CloudWatch
- Logging DAX operations using AWS CloudTrail

Monitoring tools for DynamoDB Accelerator

AWS provides tools that you can use to monitor Amazon DynamoDB Accelerator (DAX). You can configure some of these tools to do the monitoring for you, and some require manual intervention. We recommend that you automate monitoring tasks as much as possible.

Topics

- Automated monitoring tools
- Manual monitoring tools

Automated monitoring tools

You can use the following automated monitoring tools to watch DAX and report when something is wrong:

- **Amazon CloudWatch Alarms** – Watch a single metric over a time period that you specify, and perform one or more actions based on the value of the metric relative to a given threshold over

a number of time periods. The action is a notification sent to an Amazon Simple Notification Service (Amazon SNS) topic or Amazon EC2 Auto Scaling policy. CloudWatch alarms do not invoke actions simply because they are in a particular state; the state must have changed and been maintained for a specified number of periods. For more information, see Monitoring metrics in DynamoDB with Amazon CloudWatch.

- **Amazon CloudWatch Logs** – Monitor, store, and access your log files from AWS CloudTrail or other sources. For more information, see Monitoring Log Files in the *Amazon CloudWatch User Guide*.
- **Amazon CloudWatch Events** – Match events and route them to one or more target functions or streams to make changes, capture state information, and take corrective action. For more information, see What is Amazon CloudWatch Events in the *Amazon CloudWatch User Guide*.
- **AWS CloudTrail Log Monitoring** – Share log files between accounts, monitor CloudTrail log files in real time by sending them to CloudWatch Logs, write log processing applications in Java, and validate that your log files have not changed after delivery by CloudTrail. For more information, see Working with CloudTrail Log Files in the *AWS CloudTrail User Guide*.

Manual monitoring tools

Another important part of monitoring DAX involves manually monitoring those items that the CloudWatch alarms don't cover. The DAX, CloudWatch, Trusted Advisor, and other AWS Management Console dashboards provide an at-a-glance view of the state of your AWS environment. We recommend that you also check the log files on DAX.

- The DAX dashboard shows the following:
  - Service health
- The CloudWatch home page shows the following:
  - Current alarms and status
  - Graphs of alarms and resources
  - Service health status

  In addition, you can use CloudWatch to do the following:
  - Create customized dashboards to monitor the services that you care about.
  - Graph metric data to troubleshoot issues and discover trends.
  - Search and browse all of your AWS resource metrics.
  - Create and edit alarms to be notified of problems.

Monitoring with Amazon CloudWatch

You can monitor DynamoDB Accelerator (DAX) using Amazon CloudWatch, which collects and processes raw data from DAX into readable, near real-time metrics. These statistics are recorded for a period of two weeks. You can then access historical information for a better perspective on how your web application or service is performing. By default, DAX metric data is sent to CloudWatch automatically. For more information, see What Is Amazon CloudWatch? in the *Amazon CloudWatch User Guide*.

Topics

- How do I use DAX metrics?
- Viewing DAX metrics and dimensions
- Creating CloudWatch alarms to monitor DAX
- Production monitoring

How do I use DAX metrics?

The metrics reported by DAX provide information that you can analyze in different ways. The following list shows some common uses for the metrics. These are suggestions to get you started, and not a comprehensive list.

How Can I?	Relevant Metrics
Determine if any system errors occurred	Monitor `FaultRequestCount` to determine if any requests resulted in an HTTP 500 (server error) code. This can indicate a DAX internal service error or an HTTP 500 in the underlying table's SystemErrors metric.
Determine if any user errors occurred	Monitor `ErrorRequestCount` to determine if any requests resulted in an HTTP 400 (client error) code. If you see the error count growing, you might want to investigate and make sure you are sending correct client requests.
Determine if any cache misses occurred	Monitor `ItemCacheMisses` to determine the number of times an item was not found in the cache, and `QueryCach`

How Can I?	Relevant Metrics
	`eMisses` and `ScanCacheMisses` to determine the number of times a query or scan result was not found in the cache.
Monitor cache hit rates	Use CloudWatch Metric Math to define a cache hit rate metric using math expressions. For example, for the item cache, you can use the expression m1/SUM([m1, m2])*100, where m1 is the `ItemCacheHits` metric and m2 is the `ItemCacheMisses` metric for your cluster. For the query and scan caches, you can follow the same pattern using the corresponding query and scan cache metric.

Viewing DAX metrics and dimensions

When you interact with Amazon DynamoDB, it sends metrics and dimensions to Amazon CloudWatch. You can use the following procedures to view the metrics for DynamoDB Accelerator (DAX).

To view metrics (console)

Metrics are grouped first by the service namespace, and then by the various dimension combinations within each namespace.

1. Open the CloudWatch console at https://console.aws.amazon.com/cloudwatch/.
2. In the navigation pane, choose **Metrics**.
3. Select the **DAX** namespace.

To view metrics (AWS CLI)

- At a command prompt, use the following command.

```
aws cloudwatch list-metrics --namespace "AWS/DAX"
```

DAX metrics and dimensions

The following sections contain the metrics and dimensions that DAX sends to CloudWatch.

DAX Metrics

The following metrics are available from DAX. DAX sends metrics to CloudWatch only when they have a non-zero value.

> **Note**
>
> CloudWatch aggregates the following DAX metrics at one-minute intervals:
>
> - CPUUtilization
> - CacheMemoryUtilization
> - NetworkBytesIn
> - NetworkBytesOut
> - BaselineNetworkBytesInUtilization
> - BaselineNetworkBytesOutUtilization
> - NetworkPacketsIn
> - NetworkPacketsOut
> - GetItemRequestCount
> - BatchGetItemRequestCount
> - BatchWriteItemRequestCount
> - DeleteItemRequestCount
> - PutItemRequestCount
> - UpdateItemRequestCount
> - TransactWriteItemsCount
> - TransactGetItemsCount
> - ItemCacheHits
> - ItemCacheMisses
> - QueryCacheHits
> - QueryCacheMisses
> - ScanCacheHits
> - ScanCacheMisses
> - TotalRequestCount
> - ErrorRequestCount

- FaultRequestCount
- FailedRequestCount
- QueryRequestCount
- ScanRequestCount
- ClientConnections
- EstimatedDbSize
- EvictedSize
- CPUCreditUsage
- CPUCreditBalance
- CPUSurplusCreditBalance
- CPUSurplusCreditsCharged

Not all statistics, such as Average or Sum, are applicable for every metric. However, all of these values are available through the DAX console, or by using the CloudWatch console, AWS CLI, or AWS SDKs for all metrics. In the following table, each metric has a list of valid statistics that are applicable to that metric.

Metric	Description
CPUUtilization	The percentage of CPU utilization of the node or cluster. Units: Percent Valid Statistics: • Minimum • Maximum • Average
CacheMemoryUtilization	The percentage of available cache memory that is in use by the item cache and query cache on the node or cluster. Cached data starts to be evicted prior to memory utilization reaching 100% (see EvictedSize metric). If CacheMemoryUtilization reaches 100% on any node, write

Metric	Description
	requests will be throttled and you should consider switching to a cluster with a larger node type. Units: `Percent` Valid Statistics: • `Minimum` • `Maximum` • `Average`
`NetworkBytesIn`	The number of bytes received on all network interfaces by the node or cluster. Units: `Bytes` Valid Statistics: • `Minimum` • `Maximum` • `Average`
`NetworkBytesOut`	The number of bytes sent out on all network interfaces by the node or cluster. This metric identifies the volume of outgoing traffic in terms of the number of bytes on a single node or cluster. Units: `Bytes` Valid Statistics: • `Minimum` • `Maximum` • `Average`

Metric	Description
BaselineNetworkBytesInUtilization	The percentage of the consumed baseline network bandwidth at a given time for ingress traffic. For reference, 50% means half of the available network bandwidth for ingress traffic is being used. Units: Percent Valid Statistics: • Minimum • Maximum • Average • SampleCount • Sum
BaselineNetworkBytesOutUtilization	The percentage of the consumed baseline network bandwidth at a given time for egress traffic. For reference, 50% means half of the available network bandwidth for egress traffic is being used. Units: Percent Valid Statistics: • Minimum • Maximum • Average • SampleCount • Sum

Metric	Description
NetworkPacketsIn	The number of packets received on all network interfaces by the node or cluster. Units: Count Valid Statistics: • Minimum • Maximum • Average
NetworkPacketsOut	The number of packets sent out on all network interfaces by the node or cluster. This metric identifies the volume of outgoing traffic in terms of the number of packets on a single node or cluster. Units: Count Valid Statistics: • Minimum • Maximum • Average
GetItemRequestCount	The number of GetItem requests handled by the node or cluster. Units: Count Valid Statistics: • Minimum • Maximum • Average • SampleCount • Sum

Metric	Description
BatchGetItemRequestCount	The number of BatchGetItem requests handled by the node or cluster. Units: Count Valid Statistics: • Minimum • Maximum • Average • SampleCount • Sum
BatchWriteItemRequestCount	The number of BatchWriteItem requests handled by the node or cluster. Units: Count Valid Statistics: • Minimum • Maximum • Average • SampleCount • Sum

Metric	Description
DeleteItemRequestCount	The number of DeleteItem requests handled by the node or cluster. Units: Count Valid Statistics: • Minimum • Maximum • Average • SampleCount • Sum
PutItemRequestCount	The number of PutItem requests handled by the node or cluster. Units: Count Valid Statistics: • Minimum • Maximum • Average • SampleCount • Sum

Metric	Description
UpdateItemRequestCount	The number of UpdateItem requests handled by the node or cluster. Units: Count Valid Statistics: • Minimum • Maximum • Average • SampleCount • Sum
TransactWriteItemsCount	The number of TransactWriteItems requests handled by the node or cluster. Units: Count Valid Statistics: • Minimum • Maximum • Average • SampleCount • Sum

Metric	Description
TransactGetItemsCount	The number of TransactGetItems requests handled by the node or cluster. Units: Count Valid Statistics: • Minimum • Maximum • Average • SampleCount • Sum
ItemCacheHits	The number of times an item was returned from the cache by the node or cluster. Units: Count Valid Statistics: • Minimum • Maximum • Average • SampleCount • Sum

Metric	Description
ItemCacheMisses	The number of times an item was not in the node or cluster cache, and had to be retrieved from DynamoDB. Units: Count Valid Statistics: • Minimum • Maximum • Average • SampleCount • Sum
QueryCacheHits	The number of times a query result was returned from the node or cluster cache. Units: Count Valid Statistics: • Minimum • Maximum • Average • SampleCount • Sum

Metric	Description
QueryCacheMisses	The number of times a query result was not in the node or cluster cache, and had to be retrieved from DynamoDB. Units: Count Valid Statistics: • Minimum • Maximum • Average • SampleCount • Sum
ScanCacheHits	The number of times a scan result was returned from the node or cluster cache. Units: Count Valid Statistics: • Minimum • Maximum • Average • SampleCount • Sum

Metric	Description
ScanCacheMisses	The number of times a scan result was not in the node or cluster cache, and had to be retrieved from DynamoDB. Units: Count Valid Statistics: • Minimum • Maximum • Average • SampleCount • Sum
TotalRequestCount	Total number of requests handled by the node or cluster. Units: Count Valid Statistics: • Minimum • Maximum • Average • SampleCount • Sum

Metric	Description
ErrorRequestCount	Total number of requests that resulted in a user error reported by the node or cluster. Requests that were throttled by the node or cluster are included. Units: Count Valid Statistics: • Minimum • Maximum • Average • SampleCount • Sum
ThrottledRequestCount	Total number of requests throttled by the node or cluster. Requests that were throttled by DynamoDB are not included, and can be monitored using DynamoDB Metrics. Units: Count Valid Statistics: • Minimum • Maximum • Average • SampleCount • Sum

Metric	Description
FaultRequestCount	Total number of requests that resulted in an internal error reported by the node or cluster. Units: Count Valid Statistics: • Minimum • Maximum • Average • SampleCount • Sum
FailedRequestCount	Total number of requests that resulted in an error reported by the node or cluster. Units: Count Valid Statistics: • Minimum • Maximum • Average • SampleCount • Sum

Metric	Description
QueryRequestCount	The number of query requests handled by the node or cluster. Units: Count Valid Statistics: • Minimum • Maximum • Average • SampleCount • Sum
ScanRequestCount	The number of scan requests handled by the node or cluster. Units: Count Valid Statistics: • Minimum • Maximum • Average • SampleCount • Sum

Metric	Description
ClientConnections	The number of simultaneous connections made by clients to the node or cluster. Units: Count Valid Statistics: • Minimum • Maximum • Average • SampleCount • Sum
EstimatedDbSize	An approximation of the amount of data cached in the item cache and the query cache by the node or cluster. Units: Bytes Valid Statistics: • Minimum • Maximum • Average

Metric	Description
EvictedSize	The amount of data that was evicted by the node or cluster to make room for newly requested data. If the miss rate goes up, and you see this metric also growing, it probably means that your working set has increased. You should consider switching to a cluster with a larger node type. Units: Bytes Valid Statistics: • Minimum • Maximum • Average • Sum
CPUCreditUsage	The number of CPU credits spent by the node for CPU utilization. One CPU credit equals one vCPU running at 100% utilization for one minute or an equivalent combination of vCPUs, utilization, and time (for example, one vCPU running at 50% utilization for two minutes or two vCPUs running at 25% utilization for two minutes). CPU credit metrics are available at a five-minute frequency only. If you specify a period greater than five minutes, use the Sum statistic instead of the Average. Units: Credits (vCPU-minutes) Valid Statistics: • Minimum • Maximum • Average • SampleCount • Sum

Metric	Description
CPUCreditBalance	The number of earned CPU credits that a node has accrued since it was launched or started. Credits are accrued in the credit balance after they are earned, and removed from the credit balance when they are spent. The credit balance has a maximum limit, determined by the DAX node size. After the limit is reached, any new credits that are earned are discarded. The credits in the CPUCreditBalance are available for the node to spend to burst beyond its baseline CPU utilization. Units: Credits (vCPU-minutes) Valid Statistics: • Minimum • Maximum • Average • SampleCount • Sum

Metric	Description
CPUSurplusCreditBalance	The number of surplus credits that have been spent by a DAX node when its CPUCreditBalance value is zero. The CPUSurplusCreditBalance value is paid down by earned CPU credits. If the number of surplus credits exceeds the maximum number of credits that the node can earn in a 24-hour period, the spent surplus credits above the maximum incur an additional charge. Units: Credits (vCPU-minutes) Valid Statistics: • Minimum • Maximum • Average • SampleCount • Sum

Metric	Description
CPUSurplusCreditsCharged	The number of spent surplus credits that are not paid down by earned CPU credits, and which thus incur an additional charge. Spent surplus credits are charged when the spent surplus credits exceed the maximum number of credits that the node can earn in a 24-hour period. Spent surplus credits above the maximum are charged at the end of the hour or when the node is terminated. Units: Credits (vCPU-minutes) Valid Statistics: • Minimum • Maximum • Average • SampleCount • Sum

Note

The CPUCreditUsage, CPUCreditBalance, CPUSurplusCreditBalance, and CPUSurplusCreditsCharged metrics are available only for T3 nodes.

Dimensions for DAX Metrics

The metrics for DAX are qualified by the values for the account, cluster ID, or cluster ID and node ID combination. You can use the CloudWatch console to retrieve DAX data along any of the dimensions in the following table.

Dimension	Description
`Account`	Provides aggregated statistics across all nodes in an account.
`ClusterId`	Limits the data to a cluster.
`ClusterId, NodeId`	Limits the data to a node within a cluster.

Creating CloudWatch alarms to monitor DAX

You can create an Amazon CloudWatch alarm that sends an Amazon Simple Notification Service (Amazon SNS) message when the alarm changes state. An alarm watches a single metric over a time period that you specify. It performs one or more actions based on the value of the metric relative to a given threshold over a number of time periods. The action is a notification that is sent to an Amazon SNS topic or Auto Scaling policy. Alarms invoke actions for sustained state changes only. CloudWatch alarms do not invoke actions simply because they are in a particular state. The state must have changed and been maintained for a specified number of periods.

How can I be notified of query cache misses?

1. Create an Amazon SNS topic, `arn:aws:sns:us-west-2:522194210714:QueryMissAlarm`.

   For more information, see Set Up Amazon Simple Notification Service in the *Amazon CloudWatch User Guide*.

2. Create the alarm.

```
aws cloudwatch put-metric-alarm \
    --alarm-name QueryCacheMissesAlarm  \
    --alarm-description "Alarm over query cache misses" \
    --namespace AWS/DAX \
```

```
    --metric-name QueryCacheMisses  \
    --dimensions Name=ClusterID,Value=myCluster \
    --statistic Sum \
    --threshold 8 \
    --comparison-operator GreaterThanOrEqualToThreshold \
    --period 60 \
    --evaluation-periods 1 \
    --alarm-actions arn:aws:sns:us-west-2:522194210714:QueryMissAlarm
```

3. Test the alarm.

```
aws cloudwatch set-alarm-state --alarm-name QueryCacheMissesAlarm --state-reason
 "initializing" --state-value OK
```

```
aws cloudwatch set-alarm-state --alarm-name QueryCacheMissesAlarm --state-reason
 "initializing" --state-value ALARM
```

> **Note**
>
> You can increase or decrease the threshold to one that makes sense for your application. You can also use CloudWatch Metric Math to define a cache miss rate metric and set an alarm over that metric.

How can I be notified if requests cause an internal error in the cluster?

1. Create an Amazon SNS topic, `arn:aws:sns:us-west-2:123456789012:notify-on-system-errors`.

   For more information, see Set Up Amazon Simple Notification Service in the *Amazon CloudWatch User Guide*.

2. Create the alarm.

```
aws cloudwatch put-metric-alarm \
    --alarm-name FaultRequestCountAlarm \
    --alarm-description "Alarm when a request causes an internal error" \
    --namespace AWS/DAX \
    --metric-name FaultRequestCount \
    --dimensions Name=ClusterID,Value=myCluster \
```

```
    --statistic Sum \
    --threshold 0 \
    --comparison-operator GreaterThanThreshold \
    --period 60 \
    --unit Count \
    --evaluation-periods 1 \
    --alarm-actions arn:aws:sns:us-east-1:123456789012:notify-on-system-errors
```

3. Test the alarm.

```
aws cloudwatch set-alarm-state --alarm-name FaultRequestCountAlarm --state-reason
 "initializing" --state-value OK
```

```
aws cloudwatch set-alarm-state --alarm-name FaultRequestCountAlarm --state-reason
 "initializing" --state-value ALARM
```

Production monitoring

You should establish a baseline for normal DAX performance in your environment, by measuring performance at various times and under different load conditions. As you monitor DAX, you should consider storing historical monitoring data. This stored data gives you a baseline from which to compare current performance data, identify normal performance patterns and performance anomalies, and devise methods to address issues.

To establish a baseline, you should, at minimum, monitor the following items both during load testing and in production.

- CPU utilization and throttled requests, so that you can determine whether you might need to use a larger node type in your cluster. The CPU utilization of your cluster is available through the `CPUUtilization` CloudWatch metric. The average stat on this metric provides an average CPU utilization view across all the nodes in your cluster. For cluster scaling decisions, we recommend that you use the maximum stat which is the maximum utilization across all the nodes.

> **Note**
>
> AWS has improved the `CPUUtilization` metric's granularity. You might observe changes to the metric starting from 2024-05-17 to 2024-06-22.

- Operation latency (as measure on the client side) should remain consistent within your application's latency requirements.
- Error rates should remain low, as seen from the `ErrorRequestCount`, `FaultRequestCount`, and `FailedRequestCount` CloudWatch metrics.
- Network bytes consumption, so that you can determine if you should use more nodes or a larger node type in your cluster. To monitor consumption, you can set alerts on `BaselineNetworkBytesInUtilization` and `BaselineNetworkBytesOutUtilization` metrics available in CloudWatch, which indicates percentage consumption of available network bandwidth for your instance type, for ingress and egress traffic respectively.
- Cache memory utilization and evicted size, so that you can determine whether the cluster's node type has sufficient memory to hold your working set, and if not, switch to a larger node type.

  > **Note**
  >
  > In case of a large number of cache misses and writes, cache memory utilization can increase up to 100% and may cause availability downtime.

- Client connections, so that you can monitor for any unexplained spikes in connections to the cluster.

Logging DAX operations using AWS CloudTrail

Amazon DynamoDB Accelerator (DAX) is integrated with AWS CloudTrail, a service that provides a record of actions taken by a user, role, or an AWS service in DAX.

To learn more about DAX and CloudTrail, see the DynamoDB Accelerator (DAX) section in Logging DynamoDB operations by using AWS CloudTrail.

DAX T3/T2 burstable instances

DAX allows you to choose between fixed performance instances (such as R4, R5, and R7) and burstable performance instances (such as T2 and T3). Burstable performance instances provide a baseline level of CPU performance with the ability to burst above the baseline when needed.

Baseline performance and the ability to burst above it are governed by CPU credits. Burstable performance instances accumulate CPU credits continuously, at a rate determined by the instance size, when the workload is below the baseline threshold. These credits may then be consumed

when the workload increases. A CPU credit provides the performance of a full CPU core for one minute.

Many workloads don't need consistently high levels of CPU, but benefit significantly from having full access to very fast CPUs when they need them. Burstable performance instances are engineered specifically for these use cases. If you need consistently high CPU performance for your database, we recommend you use fixed performance instances.

DAX T2 instance family

DAX T2 instances are burstable general-purpose performance instances that provide a baseline level of CPU performance with the ability to burst above the baseline. T2 instances are a good choice for test and development workloads which need price predictability. DAX T2 instances are configured for standard mode, which means that if the instance is running low on accrued credits, CPU utilization is gradually lowered to the baseline level. For more information on standard mode, refer to Standard mode for burstable performance instances in the Amazon EC2 User Guide.

DAX T3 instance family

DAX T3 instances are the next generation burstable general-purpose instance type, providing a baseline level of CPU performance with the ability to burst CPU usage at any time for as long as required. T3 instances offer a balance of compute, memory, and network resources and are ideal for workloads with moderate CPU usage that experience temporary spikes in use. DAX T3 instances are configured for unlimited mode, which means they can burst beyond the baseline over a 24-hour window for an additional charge. For more information on unlimited mode, refer to Unlimited mode for burstable performance instances in the Amazon EC2 User Guide.

DAX T3 instances can sustain high CPU performance for as long as a workload requires it. For most general-purpose workloads, T3 instances will provide ample performance without any additional charges. The hourly T3 instance price automatically covers all interim spikes in usage when the average CPU utilization of a T3 instance is at or less than the baseline over a 24-hour window.

For example, a `dax.t3.small` instance receives credits continuously at a rate of 24 CPU credits per hour. This capability provides baseline performance equivalent to 20% of a CPU core (20% × 60 minutes = 12 minutes). If the instance does not use the credits it receives, they are stored in its CPU credit balance up to a maximum of 576 CPU credits. When the `t3.small` instance needs to burst to more than 20% of a core, it draws from its CPU credit balance to handle this surge automatically.

While DAX T2 instances are restricted to baseline performance once the CPU credit balance is drawn down to zero, DAX T3 instances can burst above the baseline even when their CPU credit balance is zero. For the vast majority of workloads, where the average CPU utilization is at or below the baseline performance, the basic hourly price for `t3.small` covers all CPU bursts. If the instance happens to run at an average 25% CPU utilization (5% above baseline) over a period of 24 hours after its CPU credit balance is drawn to zero, it will be charged an additional 11.52 cents (9.6 cents/vCPU-hour × 1 vCPU × 5% × 24 hours). See Amazon DynamoDB Pricing for pricing details.

DAX access control

DynamoDB Accelerator (DAX) is designed to work together with DynamoDB, to seamlessly add a caching layer to your applications. However, DAX and DynamoDB have separate access control mechanisms. Both services use AWS Identity and Access Management (IAM) to implement their respective security policies, but the security models for DAX and DynamoDB are different.

We highly recommend that you understand both security models, so that you can implement proper security measures for your applications that use DAX.

This section describes the access control mechanisms provided by DAX and provides sample IAM policies that you can tailor to your needs.

With DynamoDB, you can create IAM policies that limit the actions a user can perform on individual DynamoDB resources. For example, you can create a user role that only allows the user to perform read-only actions on a particular DynamoDB table. (For more information, see Identity and Access Management for Amazon DynamoDB.) By comparison, the DAX security model focuses on cluster security, and the ability of the cluster to perform DynamoDB API actions on your behalf.

Warning

If you are currently using IAM roles and policies to restrict access to DynamoDB tables data, then the use of DAX can **subvert** those policies. For example, a user could have access to a DynamoDB table via DAX but not have explicit access to the same table accessing DynamoDB directly. For more information, see Identity and Access Management for Amazon DynamoDB.

DAX does not enforce user-level separation on data in DynamoDB. Instead, users inherit the permissions of the DAX cluster's IAM policy when they access that cluster. Thus, when accessing DynamoDB tables via DAX, the only access controls that are in effect are the permissions in the DAX cluster's IAM policy. No other permissions are recognized.

If you require isolation, we recommend that you create additional DAX clusters and scope the IAM policy for each cluster accordingly. For example, you could create multiple DAX clusters and allow each cluster to access only a single table.

IAM service role for DAX

When you create a DAX cluster, you must associate the cluster with an IAM role. This is known as the *service role* for the cluster.

Suppose that you wanted to create a new DAX cluster named *DAXCluster01*. You could create a service role named *DAXServiceRole*, and associate the role with *DAXCluster01*. The policy for *DAXServiceRole* would define the DynamoDB actions that *DAXCluster01* could perform, on behalf of the users who interact with *DAXCluster01*.

When you create a service role, you must specify a trust relationship between *DAXServiceRole* and the DAX service itself. A trust relationship determines which entities can assume a role and make use of its permissions. The following is an example trust relationship document for *DAXServiceRole*:

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
                "Service": "dax.amazonaws.com"
            },
            "Action": "sts:AssumeRole"
        }
    ]
}
```

This trust relationship allows a DAX cluster to assume *DAXServiceRole* and perform DynamoDB API calls on your behalf.

The DynamoDB API actions that are allowed are described in an IAM policy document, which you attach to *DAXServiceRole*. The following is an example policy document.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "DaxAccessPolicy",
            "Effect": "Allow",
            "Action": [
                "dynamodb:DescribeTable",
                "dynamodb:PutItem",
                "dynamodb:GetItem",
                "dynamodb:UpdateItem",
                "dynamodb:DeleteItem",
                "dynamodb:Query",
                "dynamodb:Scan",
                "dynamodb:BatchGetItem",
                "dynamodb:BatchWriteItem",
                "dynamodb:ConditionCheckItem"
            ],
            "Resource": [
                "arn:aws:dynamodb:us-west-2:123456789012:table/Books"
            ]
        }
    ]
}
```

This policy allows DAX to perform necessary DynamoDB API actions on a DynamoDB table. The `dynamodb:DescribeTable` action is required for DAX to maintain metadata about the table, and the others are read and write actions performed on items in the table. The table, named `Books`, is in the us-west-2 Region and is owned by AWS account ID `123456789012`.

> **Note**
>
> DAX supports mechanisms to prevent the confused deputy problem during cross-Service access. For more information, see The confused deputy problem in the *IAM User Guide*.

IAM policy to allow DAX cluster access

After you create a DAX cluster, you need to grant permissions to a user so that the user can access the DAX cluster.

For example, suppose that you want to grant access to *DAXCluster01* to a user named Alice. You would first create an IAM policy (*AliceAccessPolicy*) that defines the DAX clusters and DAX API actions that the recipient can access. You would then confer access by attaching this policy to user Alice.

The following policy document gives the recipient full access on *DAXCluster01*.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Action": [
                "dax:*"
            ],
            "Effect": "Allow",
            "Resource": [
                "arn:aws:dax:us-west-2:123456789012:cache/DAXCluster01"
            ]
        }
    ]
}
```

The policy document allows access to the DAX cluster, but it does not grant any DynamoDB permissions. (The DynamoDB permissions are conferred by the DAX service role.)

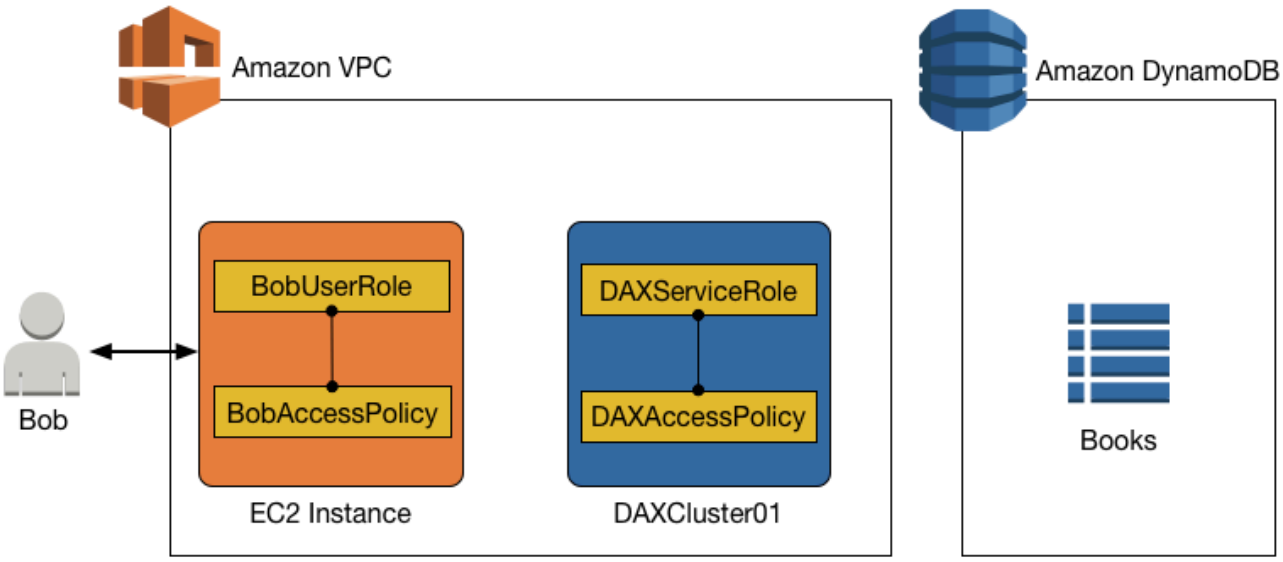
For user Alice, you would first create `AliceAccessPolicy` with the policy document shown previously. You would then attach the policy to Alice.

> **Note**
>
> Instead of attaching the policy to a user, you could attach it to an IAM role. That way, all of the users who assume that role would have the permissions that you defined in the policy.

The user policy, together with the DAX service role, determine the DynamoDB resources and API actions that the recipient can access via DAX.

Case study: Accessing DynamoDB and DAX

The following scenario can help further your understanding of IAM policies for use with DAX. (This scenario is referred to throughout the rest of this section.) The following diagram shows a high-level overview of the scenario.



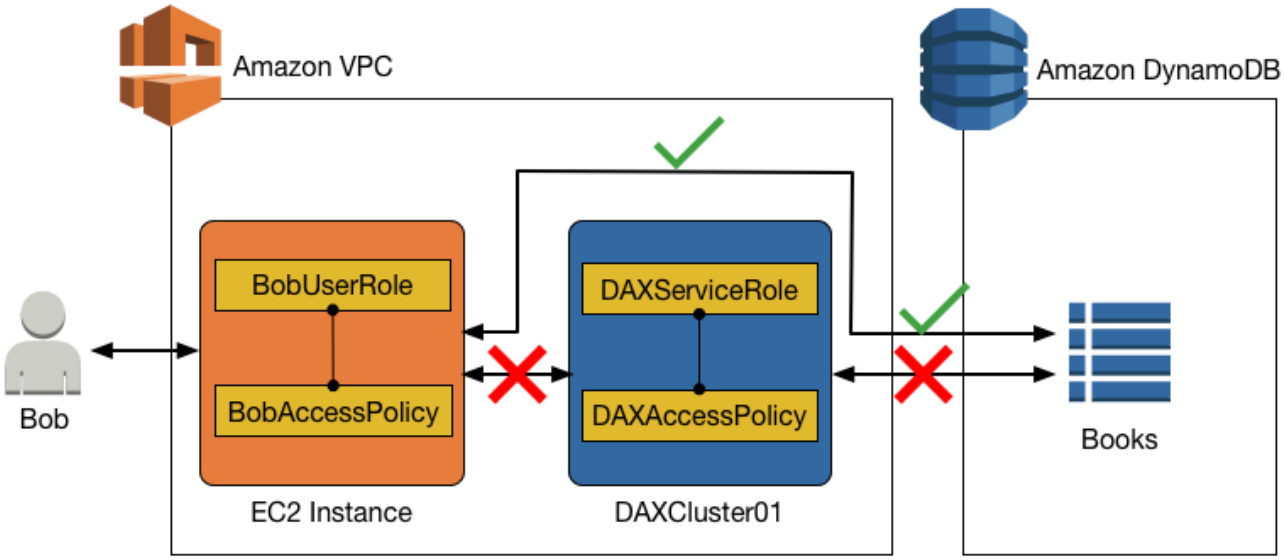
In this scenario, there are the following entities:

- A user (Bob).
- An IAM role (`BobUserRole`). Bob assumes this role at runtime.
- An IAM policy (`BobAccessPolicy`). This policy is attached to `BobUserRole`. `BobAccessPolicy` defines the DynamoDB and DAX resources that `BobUserRole` is allowed to access.
- A DAX cluster (`DAXCluster01`).
- An IAM service role (`DAXServiceRole`). This role allows `DAXCluster01` to access DynamoDB.
- An IAM policy (`DAXAccessPolicy`). This policy is attached to `DAXServiceRole`. `DAXAccessPolicy` defines the DynamoDB APIs and resources that `DAXCluster01` is allowed to access.

- A DynamoDB table (Books).

The combination of policy statements in BobAccessPolicy and DAXAccessPolicy determine what Bob can do with the Books table. For example, Bob might be able to access Books directly (using the DynamoDB endpoint), indirectly (using the DAX cluster), or both. Bob might also be able to read data from Books, write data to Books, or both.

Access to DynamoDB, but no access with DAX



It is possible to allow direct access to a DynamoDB table, while preventing indirect access using a DAX cluster. For direct access to DynamoDB, the permissions for BobUserRole are determined by BobAccessPolicy (which is attached to the role).

Read-only access to DynamoDB (only)

Bob can access DynamoDB with BobUserRole. The IAM policy attached to this role (BobAccessPolicy) determines the DynamoDB tables that BobUserRole can access, and what APIs that BobUserRole can invoke.

Consider the following policy document for BobAccessPolicy.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "DynamoDBAccessStmt",
            "Effect": "Allow",
            "Action": [
                "dynamodb:GetItem",
                "dynamodb:BatchGetItem",
                "dynamodb:Query",
                "dynamodb:Scan"
            ],
            "Resource": "arn:aws:dynamodb:us-west-2:123456789012:table/Books"
        }
    ]
}
```

When this document is attached to `BobAccessPolicy`, it allows `BobUserRole` to access the DynamoDB endpoint and perform read-only operations on the `Books` table.

DAX does not appear in this policy, so access via DAX is denied.

Read/write access to DynamoDB (only)

If `BobUserRole` requires read/write access to DynamoDB, the following policy would work.

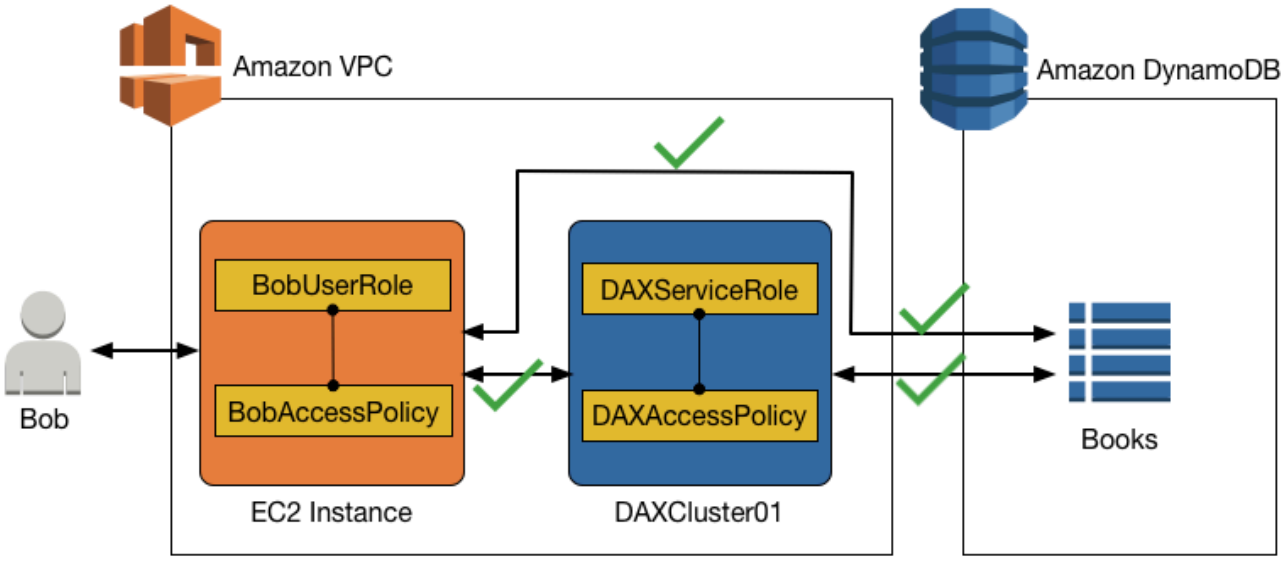
JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "DynamoDBAccessStmt",
            "Effect": "Allow",
            "Action": [
                "dynamodb:GetItem",
                "dynamodb:BatchGetItem",
                "dynamodb:Query",
                "dynamodb:Scan",
```

```
                "dynamodb:PutItem",
                "dynamodb:UpdateItem",
                "dynamodb:DeleteItem",
                "dynamodb:BatchWriteItem",
                "dynamodb:ConditionCheckItem"
            ],
            "Resource": "arn:aws:dynamodb:us-west-2:123456789012:table/Books"
        }
    ]
}
```

Again, DAX does not appear in this policy, so access via DAX is denied.

Access to DynamoDB and to DAX



To allow access to a DAX cluster, you must include DAX-specific actions in an IAM policy.

The following DAX-specific actions correspond to their similarly named counterparts in the DynamoDB API:

- `dax:GetItem`
- `dax:BatchGetItem`
- `dax:Query`

- dax:Scan
- dax:PutItem
- dax:UpdateItem
- dax:DeleteItem
- dax:BatchWriteItem
- dax:ConditionCheckItem

The same is true for the dax:EnclosingOperation condition key.

Read-only access to DynamoDB and read-only access to DAX

Suppose that Bob requires read-only access to the Books table, from DynamoDB and from DAX. The following policy (attached to BobUserRole) confers this access.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "DAXAccessStmt",
            "Effect": "Allow",
            "Action": [
                "dax:GetItem",
                "dax:BatchGetItem",
                "dax:Query",
                "dax:Scan"
            ],
            "Resource": "arn:aws:dax:us-west-2:123456789012:cache/DAXCluster01"
        },
        {
            "Sid": "DynamoDBAccessStmt",
            "Effect": "Allow",
            "Action": [
                "dynamodb:GetItem",
                "dynamodb:BatchGetItem",
                "dynamodb:Query",
                "dynamodb:Scan"
            ],
```

```
            "Resource": "arn:aws:dynamodb:us-west-2:123456789012:table/Books"
        }
    ]
}
```

The policy has a statement for DAX access (`DAXAccessStmt`) and another statement for DynamoDBaccess (`DynamoDBAccessStmt`). These statements allow Bob to send `GetItem`, `BatchGetItem`, `Query`, and `Scan` requests to `DAXCluster01`.

However, the service role for `DAXCluster01` would also require read-only access to the `Books` table in DynamoDB. The following IAM policy, attached to `DAXServiceRole`, would fulfill this requirement.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "DynamoDBAccessStmt",
            "Effect": "Allow",
            "Action": [
                "dynamodb:GetItem",
                "dynamodb:BatchGetItem",
                "dynamodb:Query",
                "dynamodb:Scan"
            ],
            "Resource": "arn:aws:dynamodb:us-west-2:123456789012:table/Books"
        }
    ]
}
```

Read/write access to DynamoDB and read-only with DAX

For a given user role, you can provide read/write access to a DynamoDB table, while also allowing read-only access via DAX.

For Bob, the IAM policy for `BobUserRole` would need to allow DynamoDB read and write actions on the `Books` table, while also supporting read-only actions via `DAXCluster01`.

The following example policy document for `BobUserRole` confers this access.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "DAXAccessStmt",
            "Effect": "Allow",
            "Action": [
                "dax:GetItem",
                "dax:BatchGetItem",
                "dax:Query",
                "dax:Scan"
            ],
            "Resource": "arn:aws:dax:us-west-2:123456789012:cache/DAXCluster01"
        },
        {
            "Sid": "DynamoDBAccessStmt",
            "Effect": "Allow",
            "Action": [
                "dynamodb:GetItem",
                "dynamodb:BatchGetItem",
                "dynamodb:Query",
                "dynamodb:Scan",
                "dynamodb:PutItem",
                "dynamodb:UpdateItem",
                "dynamodb:DeleteItem",
                "dynamodb:BatchWriteItem",
                "dynamodb:DescribeTable",
                "dynamodb:ConditionCheckItem"
            ],
            "Resource": "arn:aws:dynamodb:us-west-2:123456789012:table/Books"
        }
    ]
}
```

In addition, `DAXServiceRole` would require an IAM policy that allows `DAXCluster01` to perform read-only actions on the `Books` table.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "DynamoDBAccessStmt",
            "Effect": "Allow",
            "Action": [
                "dynamodb:GetItem",
                "dynamodb:BatchGetItem",
                "dynamodb:Query",
                "dynamodb:Scan",
                "dynamodb:DescribeTable"
           ],
            "Resource": "arn:aws:dynamodb:us-west-2:123456789012:table/Books"
        }
    ]
}
```

Read/write access to DynamoDB and read/write access to DAX

Now suppose that Bob required read/write access to the `Books` table, directly from DynamoDB or indirectly from `DAXCluster01`. The following policy document, attached to `BobAccessPolicy`, confers this access.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "DAXAccessStmt",
            "Effect": "Allow",
            "Action": [
                "dax:GetItem",
                "dax:BatchGetItem",
                "dax:Query",
                "dax:Scan",
                "dax:PutItem",
```

```
                "dax:UpdateItem",
                "dax:DeleteItem",
                "dax:BatchWriteItem",
                "dax:ConditionCheckItem"
            ],
            "Resource": "arn:aws:dax:us-west-2:123456789012:cache/DAXCluster01"
        },
        {
            "Sid": "DynamoDBAccessStmt",
            "Effect": "Allow",
            "Action": [
                "dynamodb:GetItem",
                "dynamodb:BatchGetItem",
                "dynamodb:Query",
                "dynamodb:Scan",
                "dynamodb:PutItem",
                "dynamodb:UpdateItem",
                "dynamodb:DeleteItem",
                "dynamodb:BatchWriteItem",
                "dynamodb:DescribeTable",
                "dynamodb:ConditionCheckItem"
            ],
            "Resource": "arn:aws:dynamodb:us-west-2:123456789012:table/Books"
        }
    ]
}
```

In addition, `DAXServiceRole` would require an IAM policy that allows `DAXCluster01` to perform read/write actions on the `Books` table.

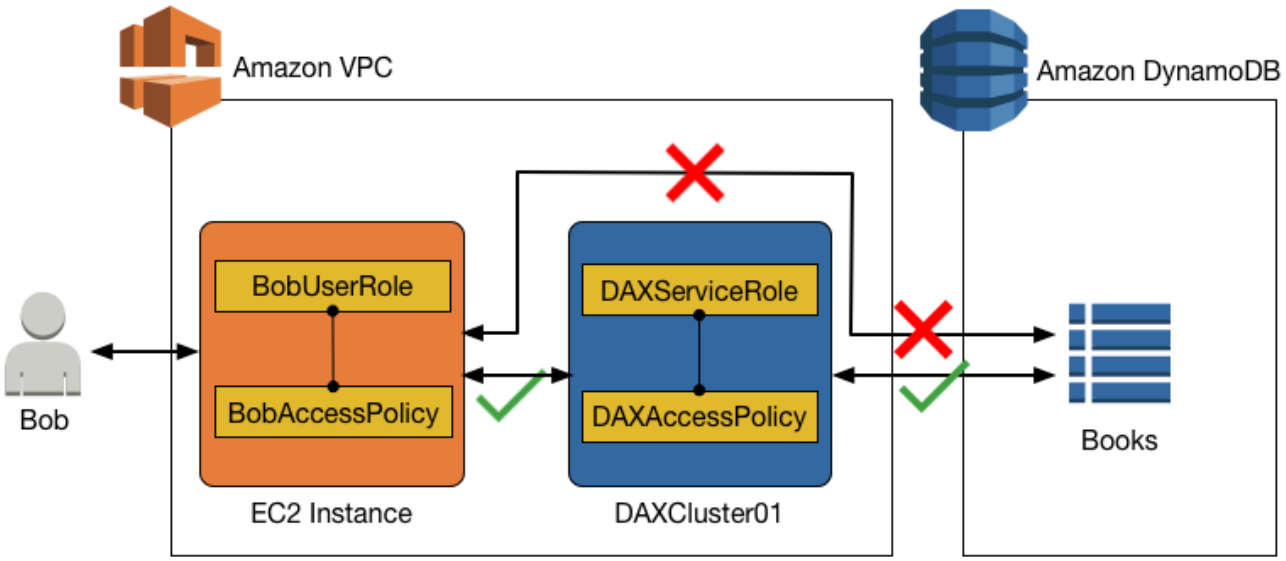
JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "DynamoDBAccessStmt",
            "Effect": "Allow",
            "Action": [
                "dynamodb:GetItem",
                "dynamodb:BatchGetItem",
```

```
                "dynamodb:Query",
                "dynamodb:Scan",
                "dynamodb:PutItem",
                "dynamodb:UpdateItem",
                "dynamodb:DeleteItem",
                "dynamodb:BatchWriteItem",
                "dynamodb:DescribeTable"
            ],
            "Resource": "arn:aws:dynamodb:us-west-2:123456789012:table/Books"
        }
    ]
}
```

Access to DynamoDB via DAX, but no direct access to DynamoDB

In this scenario, Bob can access the `Books` table via DAX, but he does not have direct access to the `Books` table in DynamoDB. Thus, when Bob gains access to DAX, he also gains access to a DynamoDB table that he otherwise might not be able to access. When you configure an IAM policy for the DAX service role, remember that any user that is given access to the DAX cluster via the user access policy gains access to the tables specified in that policy. In this case, `BobAccessPolicy` gains access to the tables specified in `DAXAccessPolicy`.



If you are currently using IAM roles and policies to restrict access to DynamoDB tables and data, using DAX can subvert those policies. In the following policy, Bob has access to a DynamoDB table via DAX but does not have explicit direct access to the same table in DynamoDB.

The following policy document (`BobAccessPolicy`), attached to `BobUserRole`, would confer this access.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "DAXAccessStmt",
            "Effect": "Allow",
            "Action": [
                "dax:GetItem",
                "dax:BatchGetItem",
                "dax:Query",
                "dax:Scan",
                "dax:PutItem",
                "dax:UpdateItem",
                "dax:DeleteItem",
                "dax:BatchWriteItem",
                "dax:ConditionCheckItem"
            ],
            "Resource": "arn:aws:dax:us-west-2:123456789012:cache/DAXCluster01"
        }
    ]
}
```

In this access policy, there are no permissions to access DynamoDB directly.

Together with `BobAccessPolicy`, the following `DAXAccessPolicy` gives `BobUserRole` access to the DynamoDB table `Books` even though `BobUserRole` cannot directly access the `Books` table.

JSON

```
{
    "Version":"2012-10-17",
```

```
    "Statement": [
        {
            "Sid": "DynamoDBAccessStmt",
            "Effect": "Allow",
            "Action": [
                "dynamodb:GetItem",
                "dynamodb:BatchGetItem",
                "dynamodb:Query",
                "dynamodb:Scan",
                "dynamodb:PutItem",
                "dynamodb:UpdateItem",
                "dynamodb:DeleteItem",
                "dynamodb:BatchWriteItem",
                "dynamodb:DescribeTable",
                "dynamodb:ConditionCheckItem"
            ],
            "Resource": "arn:aws:dynamodb:us-west-2:123456789012:table/Books"
        }
    ]
}
```

As this example shows, when you configure access control for the user access policy and the DAX cluster access policy, you must fully understand the end-to-end access to ensure that the principle of least privilege is observed. Also ensure that giving a user access to a DAX cluster does not subvert previously established access control policies.

DAX encryption at rest

Amazon DynamoDB Accelerator (DAX) encryption at rest provides an additional layer of data protection by helping secure your data from unauthorized access to the underlying storage. Organizational policies, industry or government regulations, and compliance requirements might require the use of encryption at rest to protect your data. You can use encryption to increase the data security of your applications that are deployed in the cloud.

With encryption at rest, the data persisted by DAX on disk is encrypted using 256-bit Advanced Encryption Standard, also known as AES-256 encryption. DAX writes data to disk as part of propagating changes from the primary node to read replicas.

DAX encryption at rest automatically integrates with AWS Key Management Service (AWS KMS) for managing the single service default key that is used to encrypt your clusters. If a service default

key doesn't exist when you create your encrypted DAX cluster, AWS KMS automatically creates a new AWS managed key for you. This key is used with encrypted clusters that are created in the future. AWS KMS combines secure, highly available hardware and software to provide a key management system scaled for the cloud.

After your data is encrypted, DAX handles the decryption of your data transparently with minimal impact on performance. You don't need to modify your applications to use encryption.

> **Note**
>
> DAX does not call AWS KMS for every single DAX operation. DAX only uses the key at the cluster launch. Even if access is revoked, DAX can still access the data until the cluster is shut down. Customer-specified AWS KMS keys are not supported.

DAX encryption at rest is available for the following cluster node types.

Family	Node type
Memory-optimized (R4, R5, and R7)	dax.r4.large
	dax.r4.xlarge
	dax.r4.2xlarge
	dax.r4.4xlarge
	dax.r4.8xlarge
	dax.r4.16xlarge
	dax.r5.large
	dax.r5.xlarge
	dax.r5.2xlarge
	dax.r5.4xlarge
	dax.r5.8xlarge

Family	Node type
	dax.r5.12xlarge
	dax.r5.16xlarge
	dax.r5.24xlarge
	dax.r7i.large
	dax.r7i.xlarge
	dax.r7i.2xlarge
	dax.r7i.4xlarge
	dax.r7i.8xlarge
	dax.r7i.12xlarge
	dax.r7i.16xlarge
	dax.r7i.24xlarge
General purpose (T2)	dax.t2.small
	dax.t2.medium
General purpose (T3)	dax.t3.small
	dax.t3.medium

Important

DAX encryption at rest is not supported for `dax.r3.*` node types.

You cannot enable or disable encryption at rest after a cluster has been created. You must re-create the cluster to enable encryption at rest if it was not enabled at creation.

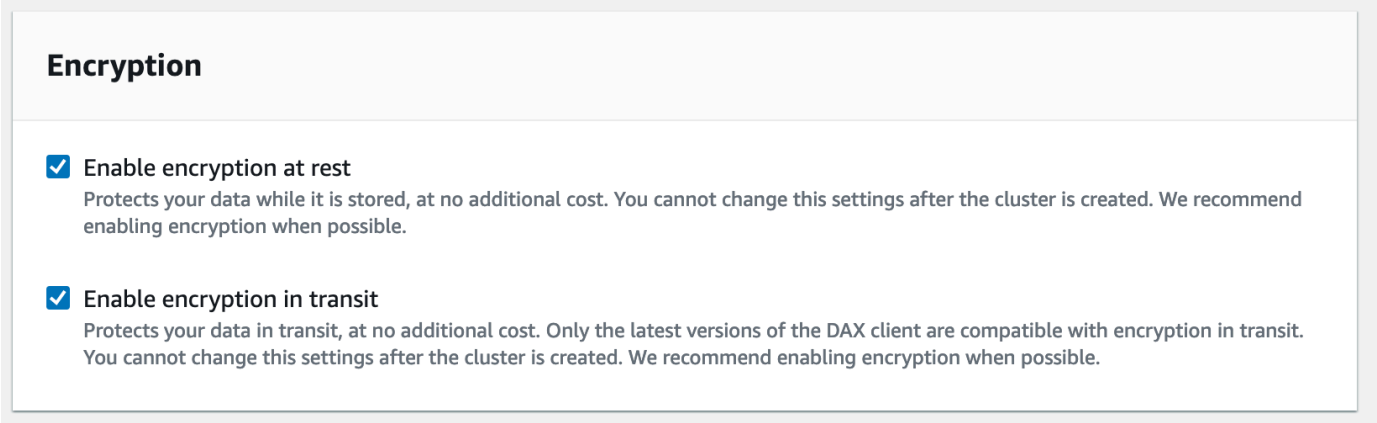
DAX encryption at rest is offered at no additional cost (AWS KMS encryption key usage charges apply). For information about pricing, see Amazon DynamoDB pricing.

Enabling encryption at rest using the AWS Management Console

Follow these steps to enable DAX encryption at rest on a table using the console.

To enable DAX encryption at rest

1. Sign in to the AWS Management Console and open the DynamoDB console at https://console.aws.amazon.com/dynamodb/.
2. In the navigation pane on the left side of the console, under **DAX**, choose **Clusters**.
3. Choose **Create cluster**.
4. For **Cluster name**, enter a short name for your cluster. Choose the **node type** for all of the nodes in the cluster, and for the cluster size, use **3** nodes.
5. In **Encryption**, make sure that **Enable encryption** is selected.



6. After choosing the IAM role, subnet group, security groups, and cluster settings, choose **Launch cluster**.

To confirm that the cluster is encrypted, check the cluster details under the **Clusters** pane. Encryption should be **ENABLED**.

DAX encryption in transit

Amazon DynamoDB Accelerator (DAX) supports encryption in transit of data between your application and your DAX cluster, enabling you to use DAX in applications with stringent encryption requirements.

Regardless of whether or not you choose encryption in transit, traffic between your application and your DAX cluster remains in your Amazon VPC. This traffic is routed to Elastic Network Interfaces with private IPs in your VPC that are attached to the nodes of your cluster. With your VPC as the trust boundary, you have significant control over the security of your data through the use of standard tools like security groups, subnet segmentation with Network ACLs, and VPC flow tracing. DAX encryption in transit adds to this baseline level of confidentiality, ensuring that all requests and responses between the application and the cluster are encrypted by transport level security (TLS), and connections to the cluster can be authenticated by verification of a cluster x509 certificate. Data written to disk by DAX can also be encrypted if you choose encryption at rest when creating your DAX cluster.

Using encryption in transit with DAX is easy. Simply select this option when creating a new cluster, and use a recent version of any of the DAX clients in your application. Clusters that use encryption in transit do not support unencrypted traffic, so there is no chance to misconfigure your application and bypass encryption. The DAX client will use the cluster's x509 certificate to authenticate the identity of the cluster when it establishes connections, ensuring that your DAX requests go where intended. All methods of creating DAX clusters support encryption in transit: the AWS Management Console, AWS CLI, all SDKs, and CloudFormation.

Encryption in transit cannot be enabled on an existing DAX cluster. To use encryption in transit in an existing DAX application, create a new cluster with encryption in transit enabled, shift your application's traffic to it, then delete the old cluster.

Using service-linked IAM roles for DAX

Amazon DynamoDB Accelerator (DAX) uses AWS Identity and Access Management (IAM) service-linked roles. A service-linked role is a unique type of IAM role that is linked directly to DAX. Service-linked roles are predefined by DAX and include all the permissions that the service requires to call other AWS services on your behalf.

A service-linked role makes setting up DAX easier because you don’t have to manually add the necessary permissions. DAX defines the permissions of its service-linked roles, and unless defined otherwise, only DAX can assume its roles. The defined permissions include the trust policy and the permissions policy. That permissions policy can't be attached to any other IAM entity.

You can delete the roles only after first deleting their related resources. This protects your DAX resources because you can't inadvertently remove permission to access the resources.

For information about other services that support service-linked roles, see AWS Services That Work with IAM in the *IAM User Guide*. Look for the services that have **Yes** in the **Service-linked roles** column. Choose a **Yes** link to view the service-linked role documentation for that service.

Topics

- Service-linked role permissions for DAX
- Creating a service-linked role for DAX
- Editing a service-linked role for DAX
- Deleting a service-linked role for DAX

Service-linked role permissions for DAX

DAX uses the service-linked role named `AWSServiceRoleForDAX`. This role allows DAX to call services on behalf of your DAX cluster.

> **Important**
>
> The `AWSServiceRoleForDAX` service-linked role makes it easier for you to set up and maintain a DAX cluster. However, you must still grant each cluster access to DynamoDB before you can use it. For more information, see DAX access control.

The `AWSServiceRoleForDAX` service-linked role trusts the following services to assume the role:

- `dax.amazonaws.com`

The role permissions policy allows DAX to complete the following actions on the specified resources:

- Actions on `ec2`:
  - `AuthorizeSecurityGroupIngress`
  - `CreateNetworkInterface`
  - `CreateSecurityGroup`
  - `DeleteNetworkInterface`
  - `DeleteSecurityGroup`

- `DescribeAvailabilityZones`
- `DescribeNetworkInterfaces`
- `DescribeSecurityGroups`
- `DescribeSubnets`
- `DescribeVpcs`
- `ModifyNetworkInterfaceAttribute`
- `RevokeSecurityGroupIngress`

You must configure permissions to allow an IAM entity (such as a user, group, or role) to create, edit, or delete a service-linked role. For more information, see Service-Linked Role Permissions in the *IAM User Guide*.

To allow an IAM entity to create AWSServiceRoleForDAX service-linked roles

Add the following policy statement to the permissions for that IAM entity.

```
{
    "Effect": "Allow",
    "Action": [
        "iam:CreateServiceLinkedRole"
    ],
    "Resource": "*",
    "Condition": {"StringLike": {"iam:AWSServiceName": "dax.amazonaws.com"}}
}
```

Creating a service-linked role for DAX

You don't need to manually create a service-linked role. When you create a cluster, DAX creates the service-linked role for you.

> **Important**
>
> If you were using the DAX service before February 28, 2018, when it began supporting service-linked roles, DAX created the `AWSServiceRoleForDAX` role in your account. For more information, see A New Role Appeared in My AWS Account in the *IAM User Guide*.

If you delete this service-linked role and then need to create it again, you can use the same process to re-create the role in your account. When you create an instance or a cluster, DAX creates the service-linked role for you again.

Editing a service-linked role for DAX

DAX does not allow you to edit the `AWSServiceRoleForDAX` service-linked role. After you create a service-linked role, you cannot change the name of the role because various entities might reference the role. However, you can edit the description of the role using IAM. For more information, see Editing a Service-Linked Role in the *IAM User Guide*.

Deleting a service-linked role for DAX

If you no longer need to use a feature or service that requires a service-linked role, we recommend that you delete that role. That way you don't have an unused entity that is not actively monitored or maintained. However, you must delete all of your DAX clusters before you can delete the service-linked role.

Cleaning up a service-linked role

Before you can use IAM to delete a service-linked role, you must first confirm that the role has no active sessions and remove any resources used by the role.

To check whether the service-linked role has an active session in the IAM console

1. Sign in to the AWS Management Console and open the IAM console at https://console.aws.amazon.com/iam/.
2. In the navigation pane of the IAM console, choose **Roles**. Then choose the name (not the check box) of the **AWSServiceRoleForDAX** role.
3. On the **Summary** page for the selected role, choose the **Access Advisor** tab.
4. On the **Access Advisor** tab, review recent activity for the service-linked role.

> **Note**
>
> If you are unsure whether DAX is using the `AWSServiceRoleForDAX` role, you can try to delete the role. If the service is using the role, the deletion fails, and you can view the Regions where the role is being used. If the role is being used, you must delete your

DAX clusters before you can delete the role. You can't revoke the session for a service-linked role.

If you want to remove the `AWSServiceRoleForDAX` role, you must first delete all of your DAX clusters.

Deleting all of your DAX clusters

Use one of these procedures to delete each of your DAX clusters.

To delete a DAX cluster (console)

1. Open the DynamoDB console at https://console.aws.amazon.com/dynamodb/.
2. In the navigation pane, under **DAX**, choose **Clusters**.
3. Choose **Actions**, and then choose **Delete**.
4. In the **Delete cluster confirmation** box, choose **Delete**.

To delete a DAX cluster (AWS CLI)

See delete-cluster in the *AWS CLI Command Reference*.

To delete a DAX cluster (API)

See DeleteCluster in the *Amazon DynamoDB API Reference*.

Deleting the service-linked role

To manually delete the service-linked role using IAM

Use the IAM console, the IAM CLI, or the IAM API to delete the `AWSServiceRoleForDAX` service-linked role. For more information, see Deleting a Service-Linked Role in the *IAM User Guide*.

Accessing DAX across AWS accounts

Imagine that you have a DynamoDB Accelerator (DAX) cluster running in one AWS account (account A), and the DAX cluster needs to be accessible from an Amazon Elastic Compute Cloud (Amazon EC2) instance in another AWS account (account B). In this tutorial, you accomplish this by launching an EC2 instance in account B with an IAM role from account B. You then use temporary security credentials from the EC2 instance to assume an IAM role from account A. Finally, you use the

temporary security credentials from assuming the IAM role in account A to make application calls over an Amazon VPC peering connection to the DAX cluster in account A. In order to perform these tasks you will need administrative access in both AWS accounts.

> **Important**
>
> It is not possible to have a DAX cluster access a DynamoDB table from a different account.

Topics

- Set up IAM
- Set up a VPC
- Modify the DAX client to allow cross-account access

Set up IAM

1. Create a text file named `AssumeDaxRoleTrust.json` with the following content, which allows Amazon EC2 to work on your behalf.

   JSON

   ```
   {
       "Version":"2012-10-17",
       "Statement": [
           {
               "Effect": "Allow",
               "Principal": {
                   "Service": "ec2.amazonaws.com"
               },
               "Action": "sts:AssumeRole"
           }
       ]
   }
   ```

2. In account B, create a role that Amazon EC2 can use when launching instances.

   ```
   aws iam create-role \
       --role-name AssumeDaxRole \
   ```

```
    --assume-role-policy-document file://AssumeDaxRoleTrust.json
```

3. Create a text file named `AssumeDaxRolePolicy.json` with the following content, which allows code running on the EC2 instance in account B to assume an IAM role in account A. Replace *accountA* with the actual ID of account A.

   JSON

   ```
   {
       "Version":"2012-10-17",
       "Statement": [
           {
               "Effect": "Allow",
               "Action": "sts:AssumeRole",
               "Resource": "arn:aws:iam::111122223333:role/DaxCrossAccountRole"
           }
       ]
   }
   ```

4. Add that policy to the role you just created.

   ```
   aws iam put-role-policy \
       --role-name AssumeDaxRole \
       --policy-name AssumeDaxRolePolicy \
       --policy-document file://AssumeDaxRolePolicy.json
   ```

5. Create an instance profile to allow instances to use the role.

   ```
   aws iam create-instance-profile \
       --instance-profile-name AssumeDaxInstanceProfile
   ```

6. Associate the role with the instance profile.

   ```
   aws iam add-role-to-instance-profile \
       --instance-profile-name AssumeDaxInstanceProfile \
       --role-name AssumeDaxRole
   ```

7. Create a text file named `DaxCrossAccountRoleTrust.json` with the following content, which allows account B to assume an account A role. Replace *accountB* with the actual ID of account B.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
                "AWS": "arn:aws:iam::111122223333:role/AssumeDaxRole"
            },
            "Action": "sts:AssumeRole"
        }
    ]
}
```

8. In account A, create the role that account B can assume.

```
aws iam create-role \
    --role-name DaxCrossAccountRole \
    --assume-role-policy-document file://DaxCrossAccountRoleTrust.json
```

9. Create a text file named `DaxCrossAccountPolicy.json` that allows access to the DAX cluster. Replace *dax-cluster-arn* with the correct Amazon Resource Name (ARN) of your DAX cluster.

   JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "dax:GetItem",
                "dax:BatchGetItem",
                "dax:Query",
                "dax:Scan",
                "dax:PutItem",
                "dax:UpdateItem",
                "dax:DeleteItem",
```

```
                "dax:BatchWriteItem",
                "dax:ConditionCheckItem"
            ],
            "Resource": "arn:aws:dax:us-east-1:111122223333:cache/dax-
cluster-name"
        }
    ]
}
```

10. In account A, add the policy to the role.

```
aws iam put-role-policy \
    --role-name DaxCrossAccountRole \
    --policy-name DaxCrossAccountPolicy \
    --policy-document file://DaxCrossAccountPolicy.json
```

Set up a VPC

1. Find the subnet group of account A's DAX cluster. Replace *cluster-name* with the name of the DAX cluster that account B must access.

```
aws dax describe-clusters \
    --cluster-name cluster-name
    --query 'Clusters[0].SubnetGroup'
```

2. Using that *subnet-group*, find the cluster's VPC.

```
aws dax describe-subnet-groups \
    --subnet-group-name subnet-group \
    --query 'SubnetGroups[0].VpcId'
```

3. Using that *vpc-id*, find the VPC's CIDR.

```
aws ec2 describe-vpcs \
    --vpc vpc-id \
    --query 'Vpcs[0].CidrBlock'
```

4. From account B, create a VPC using a different, non-overlapping CIDR than the one found in the previous step. Then, create at least one subnet. You can use either the VPC creation wizard in the AWS Management Console or the AWS CLI.

5. From account B, request a peering connection to the account A VPC as described in Creating and accepting a VPC peering connection. From account A, accept the connection.
6. From account B, find the new VPC's routing table. Replace *vpc-id* with the ID of the VPC you created in account B.

```
aws ec2 describe-route-tables \
    --filters 'Name=vpc-id,Values=vpc-id' \
    --query 'RouteTables[0].RouteTableId'
```

7. Add a route to send traffic destined for account A's CIDR to the VPC peering connection. Remember to replace each *user input placeholder* with the correct values for your accounts.

```
aws ec2 create-route \
    --route-table-id accountB-route-table-id \
    --destination-cidr accountA-vpc-cidr \
    --vpc-peering-connection-id peering-connection-id
```

8. From account A, find the DAX cluster's route table using the *vpc-id* you found previously.

```
aws ec2 describe-route-tables \
    --filters 'Name=vpc-id, Values=accountA-vpc-id' \
    --query 'RouteTables[0].RouteTableId'
```

9. From account A, add a route to send traffic destined for account B's CIDR to the VPC peering connection. Replace each *user input placeholder* with the correct values for your accounts.

```
aws ec2 create-route \
    --route-table-id accountA-route-table-id \
    --destination-cidr accountB-vpc-cidr \
    --vpc-peering-connection-id peering-connection-id
```

10. From account B, launch an EC2 instance in the VPC that you created earlier. Give it the `AssumeDaxInstanceProfile`. You can use either the launch wizard in the AWS Management Console or the AWS CLI. Take note of the instance's security group.
11. From account A, find the security group used by the DAX cluster. Remember to replace *cluster-name* with the name of your DAX cluster.

```
aws dax describe-clusters \
```

```
    --cluster-name cluster-name \
    --query 'Clusters[0].SecurityGroups[0].SecurityGroupIdentifier'
```

12. Update the DAX cluster's security group to allow inbound traffic from the security group of the EC2 instance you created in account B. Remember to replace the *user input placeholders* with the correct values for your accounts.

```
aws ec2 authorize-security-group-ingress \
    --group-id accountA-security-group-id \
    --protocol tcp \
    --port 8111 \
    --source-group accountB-security-group-id \
    --group-owner accountB-id
```

At this point, an application on account B's EC2 instance is able to use the instance profile to assume the `arn:aws:iam::`*accountA-id*`:role/DaxCrossAccountRole` role and use the DAX cluster.

Modify the DAX client to allow cross-account access

> **Note**
>
> AWS Security Token Service (AWS STS) credentials are temporary credentials. Some clients handle refreshing automatically, while others require additional logic to refresh the credentials. We recommend that you follow the guidance of the appropriate documentation.

Java

This section helps you modify your existing DAX client code to allow cross-account DAX access. If you don't have DAX client code already, you can find working code examples in the Java and DAX tutorial.

1. Add the following imports.

```
import com.amazonaws.auth.STSAssumeRoleSessionCredentialsProvider;
import com.amazonaws.services.securitytoken.AWSSecurityTokenService;
```

```
import
 com.amazonaws.services.securitytoken.AWSSecurityTokenServiceClientBuilder;
```

2. Get a credentials provider from AWS STS and create a DAX client object. Remember to replace each *user input placeholder* with the correct values for your accounts.

```
AWSSecurityTokenService awsSecurityTokenService =
 AWSSecurityTokenServiceClientBuilder
     .standard()
     .withRegion(region)
     .build();

STSAssumeRoleSessionCredentialsProvider credentials =  new
 STSAssumeRoleSessionCredentialsProvider.Builder("arn:aws:iam::accountA:role/
RoleName", "TryDax")
     .withStsClient(awsSecurityTokenService)
     .build();

DynamoDB client = AmazonDaxClientBuilder.standard()
    .withRegion(region)
    .withEndpointConfiguration(dax_endpoint)
    .withCredentials(credentials)
    .build();
```

.NET

This section helps you modify your existing DAX client code to allow cross-account DAX access. If you don't have DAX client code already, you can find working code examples in the .NET and DAX tutorial.

1. Add the AWSSDK.SecurityToken NuGet package to the solution.

```
<PackageReference Include="AWSSDK.SecurityToken" Version="latest version" />
```

2. Use the `SecurityToken` and `SecurityToken.Model` packages.

```
using Amazon.SecurityToken;
using Amazon.SecurityToken.Model;
```

3. Get temporary credentials from `AmazonSimpleTokenService` and create a `ClusterDaxClient` object. Remember to replace each *`user input placeholder`* with the correct values for your accounts.

```
IAmazonSecurityTokenService sts = new AmazonSecurityTokenServiceClient();

var assumeRoleResponse = sts.AssumeRole(new AssumeRoleRequest
{
    RoleArn = "arn:aws:iam::accountA:role/RoleName",
                RoleSessionName = "TryDax"
});

Credentials credentials = assumeRoleResponse.Credentials;

var clientConfig = new DaxClientConfig(dax_endpoint, port)
{
    AwsCredentials = assumeRoleResponse.Credentials

};

var client = new ClusterDaxClient(clientConfig);
```

Go

This section helps you modify your existing DAX client code to allow cross-account DAX access. If you don't have DAX client code already, you can find working code examples on GitHub.

1. Import the AWS STS and session packages.

```
import (
    "github.com/aws/aws-sdk-go/aws/session"
    "github.com/aws/aws-sdk-go/service/sts"
    "github.com/aws/aws-sdk-go/aws/credentials/stscreds"
)
```

2. Get temporary credentials from `AmazonSimpleTokenService` and create a DAX client object. Remember to replace each *`user input placeholder`* with the correct values for your accounts.

```
sess, err := session.NewSession(&aws.Config{
    Region: aws.String(region)},
```

```
)
if err != nil {
    return nil, err
}

stsClient := sts.New(sess)
arp := &stscreds.AssumeRoleProvider{
                Duration:     900 * time.Second,
                ExpiryWindow: 10 * time.Second,
                RoleARN:      "arn:aws:iam::accountA:role/role_name",
                Client:       stsClient,
                RoleSessionName: "session_name",
        }cfg := dax.DefaultConfig()

cfg.HostPorts = []string{dax_endpoint}
cfg.Region = region
cfg.Credentials = credentials.NewCredentials(arp)
daxClient := dax.New(cfg)
```

Python

This section helps you modify your existing DAX client code to allow cross-account DAX access. If you don't have DAX client code already, you can find working code examples in the Python and DAX tutorial.

1. Import `boto3`.

```
import boto3
```

2. Get temporary credentials from `sts` and create an `AmazonDaxClient` object. Remember to replace each *`user input placeholder`* with the correct values for your accounts.

```
sts = boto3.client('sts')
stsresponse =
 sts.assume_role(RoleArn='arn:aws:iam::accountA:role/
RoleName',RoleSessionName='tryDax')
credentials = botocore.session.get_session()['Credentials']

dax = amazondax.AmazonDaxClient(session, region_name=region,
 endpoints=[dax_endpoint], aws_access_key_id=credentials['AccessKeyId'],
 aws_secret_access_key=credentials['SecretAccessKey'],
 aws_session_token=credentials['SessionToken'])
```

```
client = dax
```

Node.js

This section helps you modify your existing DAX client code to allow cross-account DAX access. If you don't have DAX client code already, you can find working code examples in the Node.js and DAX tutorial. Remember to replace each *user input placeholder* with the correct values for your accounts.

```
const AmazonDaxClient = require('amazon-dax-client');
const AWS = require('aws-sdk');
const region = 'region';
const endpoints = [daxEndpoint1, ...];

const getCredentials = async() => {
  return new Promise((resolve, reject) => {
    const sts = new AWS.STS();
    const roleParams = {
      RoleArn: 'arn:aws:iam::accountA:role/RoleName',
      RoleSessionName: 'tryDax',
    };
    sts.assumeRole(roleParams, (err, session) => {
      if(err) {
        reject(err);
      } else {
        resolve({
          accessKeyId: session.Credentials.AccessKeyId,
          secretAccessKey: session.Credentials.SecretAccessKey,
          sessionToken: session.Credentials.SessionToken,
        });
      }
    });
  });
};

const createDaxClient = async() => {
  const credentials = await getCredentials();
  const daxClient = new AmazonDaxClient({endpoints: endpoints, region: region,
 accessKeyId: credentials.accessKeyId, secretAccessKey: credentials.secretAccessKey,
 sessionToken: credentials.sessionToken});
  return new AWS.DynamoDB.DocumentClient({service: daxClient});
};
```

```
createDaxClient().then((client) => {
  client.get(...);
  ...
}).catch((error) => {
  console.log('Caught an error: ' + error);
});
```

DAX cluster sizing guide

This guide provides advice for choosing an appropriate Amazon DynamoDB Accelerator (DAX) cluster size and node type for your application. These instructions guide you through the steps of estimating your application's DAX traffic, selecting a cluster configuration, and testing it.

If you have an existing DAX cluster and want to evaluate whether it has the appropriate number and size of nodes, please refer to Scaling a DAX cluster.

Topics

- Overview
- Estimating traffic
- Load testing

Overview

It's important to scale your DAX cluster appropriately for your workload, whether you're creating a new cluster or maintaining an existing cluster. As time goes on and your application's workload changes, you should periodically revisit your scaling decisions to make sure that they are still appropriate.

The process typically follows these steps:

1. **Estimating traffic.** In this step, you make predictions about the volume of traffic that your application will send to DAX, the nature of the traffic (read vs. write operations), and the expected cache hit rate.
2. **Load testing.** In this step, you create a cluster and send traffic to it mirroring your estimates from the previous step. Repeat this step until you find a suitable cluster configuration.

3. **Production monitoring.** While your application is using DAX in production, you should monitor the cluster to continuously validate that it is still scaled correctly as your workload changes over time.

Estimating traffic

There are three main factors that characterize a typical DAX workload:

- Cache hit rate
- Read capacity units (RCUs) per second
- Write capacity units (WCUs) per second

Estimating cache hit rate

If you already have a DAX cluster, you can use the `ItemCacheHits` and `ItemCacheMisses` Amazon CloudWatch metrics to determine the cache hit rate. The cache hit rate is equal to `ItemCacheHits` / (`ItemCacheHits` + `ItemCacheMisses`). If your workload includes `Query` or `Scan` operations, you should also look at the `QueryCacheHits`, `QueryCacheMisses`, `ScanCacheHits`, and `ScanCacheMisses` metrics. Cache hit rates vary from one application to another and are heavily influenced by the cluster's Time to Live (TTL) setting. Typical hit rates for applications using DAX are 85–95 percent.

Estimating read and write capacity units

If you already have DynamoDB tables for your application, look at the `ConsumedReadCapacityUnits` and `ConsumedWriteCapacityUnits` CloudWatch metrics. Use the `Sum` statistic and divide by the number of seconds in the period.

If you also already have a DAX cluster, remember that the DynamoDB `ConsumedReadCapacityUnits` metric only accounts for cache misses. So, to get an idea of the read capacity units per second handled by your DAX cluster, divide the number by your cache miss rate (that is, 1 - cache hit rate).

If you don't already have a DynamoDB table, see the documentation about read and write capacity units to estimate your traffic based on your application's estimated request rate, items accessed per request, and item size.

When making traffic estimates, plan for future growth and for expected and unexpected peaks to ensure that your cluster has enough headroom for traffic increases.

Load testing

The next step after estimating traffic is to test the cluster configuration under load.

1. For your initial load test, we recommend that you start with the `dax.r4.large` node type, the lowest-cost fixed performance, memory-optimized node type.
2. A fault-tolerant cluster requires at least three nodes, spread across three Availability Zones. In this case, if an Availability Zone becomes unavailable, the effective number of Availability Zones is reduced by one-third. For your initial load test, we recommend that you start with a two-node cluster, which simulates the failure of one Availability Zone in a three-node cluster.
3. Drive sustained traffic (as estimated in the previous step) to your test cluster for the duration of the load test.
4. Monitor the performance of the cluster during the load test.

Ideally, the traffic profile that you drive during the load test should be as similar as possible to your application's real traffic. This includes the distribution of operations (for example, 70 percent `GetItem`, 25 percent `Query`, and 5 percent `PutItem`), the request rate for each operation, the number of items accessed per request, and the distribution of item sizes. To achieve a cache hit rate similar to your application's expected cache hit rate, pay close attention to the distribution of keys in your test traffic.

> **Note**
>
> Be careful when load testing T2 node types (`dax.t2.small` and `dax.t2.medium`). T2 node types provide burstable CPU performance that varies over time depending on the node's CPU credit balance. A DAX cluster running on T2 nodes might appear to be operating normally, but if any node is bursting above the baseline performance of its instance, the node is spending its accrued CPU credit balance. When the credit balance runs low, performance is gradually lowered to the baseline performance level.

Monitor your DAX cluster during the load test to determine whether the node type that you're using for the load test is the right node type for you. In addition, during a load test, you should

monitor your request rate and cache hit rate to ensure that your test infrastructure is actually driving the amount of traffic you intend.

You should pay attention to network bytes consumption of your selected cluster instance type. Exceeding the available baseline bandwidth for an Amazon EC2 instance indicates that your cluster may not sustain your application's workload, and needs to be scaled.

If load testing indicates that the selected cluster configuration can't sustain your application's workload, you should switch to a larger node type, especially if you see high CPU utilization on the primary node in the cluster, high eviction rates, or high cache memory utilization. If hit rates are consistently high, and the ratio of read to write traffic is high, you may want to consider adding more nodes to your cluster. Refer to Scaling a DAX cluster for additional guidance on when to use a larger node type (vertical scaling) or add more nodes (horizontal scaling).

You should repeat your load test after making changes to your cluster configuration.

Data modeling for DynamoDB tables

Before we dive into data modeling, it's important to understand some DynamoDB fundamentals. DynamoDB is a key-value NoSQL database which allows flexible schema. The set of data attributes apart from the key attributes for each item can be either uniform or discrete. The DynamoDB key schema is in the form of either a simple primary key where a partition key uniquely identifies an item, or in the form of a composite primary key where a combination of a partition key and sort key uniquely defines an item. The partition key is hashed to determine the physical location of data and retrieve it. Therefore, it is important to choose a high cardinality and horizontally scalable attribute as a partition key to ensure even distribution of data. The sort key attribute is optional in the key schema and having a sort key enables modelling one-to-many relationships and creating item collections in DynamoDB. Sort keys are also referred to as range keys—they are used to sort items in an item collection and also allow flexible range-based operations.

For more details and best practices on DynamoDB key schema, you can refer to the following:

- the section called “Partitions and data distribution in DynamoDB”
- the section called “Partition key design”
- the section called “Sort key design”
- Choosing the right DynamoDB partition key

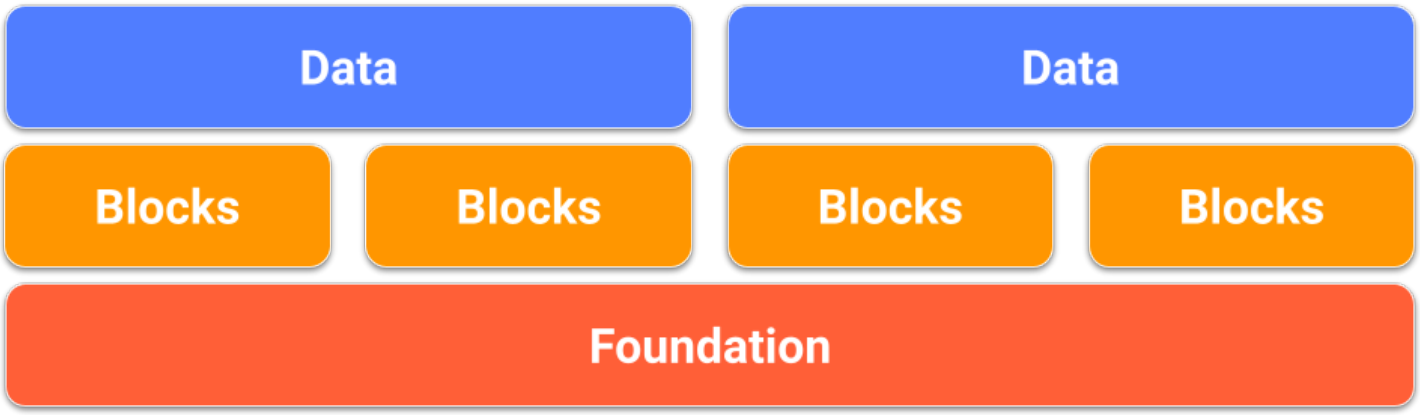
Secondary indexes are often needed to support additional query patterns in DynamoDB. Secondary indexes are shadow tables where the same data is organised via a different key schema compared to the base table. A local secondary index (LSI) shares the same partition key as the base table and allows having an alternate sort key allowing it to share the base table’s capacity. A global secondary index (GSI) can have a different partition key as well as a different sort key attribute than the base table which means throughput management for a GSI is independent of the base table.

For more details on secondary indexes and best practices, you can refer to the following:

- the section called “Working with indexes”
- the section called “Secondary indexes”

Let's now look at data modeling a little closer. The process of designing a flexible and highly-optimized schema on DynamoDB, or any NoSQL database for that matter, can be a challenging

skill to learn. The goal of this module is to help you develop a mental flowchart for designing a schema that will take you from use case into production. We will start with an introduction to the foundational choice of any design, single table versus multiple table design. Then we will review the multitude of design patterns (building blocks) that can be used to achieve various organizational or performance results for your application. Finally, we are including a variety of complete schema design packages for different use cases and industries.



Topics

- Item collections - how to model one-to-many relationships in DynamoDB
- Data Modeling foundations in DynamoDB
- Data modeling building blocks in DynamoDB
- Data modeling schema design packages in DynamoDB
- Best practices for modeling relational data in DynamoDB

Item collections - how to model one-to-many relationships in DynamoDB

In DynamoDB, an *item collection* is a group of items that share the same partition key value, which means the items are related. Item collections are the primary mechanism to model one-to-many relationships in DynamoDB. Item collections can only exist on tables or indexes configured to use a composite primary key.

> **Note**
>
> Item collections can exist either in a base table or a secondary index. For more information specifically about how item collections interact with indexes, see Item collections in Local Secondary Indexes.

Consider the following table showing three different users and their in-game inventories:

Primary key		Attributes		
Partition key: PK	Sort key: SK			
account1234	inventory::armor	data		
		{"armor": [{"name" : "Pauldron of the Paladin", "type" : "chest", "gear score" : 545 }, { "name" : "Greaves of the Ranger", "type" : "sword", "gear score" : 382}]}		
	inventory::weapons	data		
		{"weapons" : [{ "name" : "Sword of the Ancients", "type" : "sword", "gear score" : 320}]}		
	login-data	pw	state	last-login
		d1e8a70b5ccab1dc2f56bbf7e99f064a660c08e361a35751b9c483c88943d082	Active	1649276737
account1387	info	data		
		{"email" : "bot123@gmail.com"}		
	inventory::armor	data		
		{"armor": [{"name" : "Pauldron of the Paladin", "type" : "chest", "gear score" : 545 }, { "name" : "Greaves of the Ranger", "type" : "sword", "gear score" : 382}]}		
	login-data	pw	state	last-login
		k2g8jk0m5ppab1dc2f56bbf7e99f064a660c08e361a35751b9c464r23943i082	Banned	1649456737
account1138	info	data		
		{"email" : "luh-3417@gmail.com" }		
	login-data	pw	state	last-login
		88A41A9A62B11CCC8C120861928765A3EA41DEB9EAFE261D90F619473B89A2D4	Active	14275516966

For some items in each collection, the sort key is a concatenation made up of information used to group data, such as `inventory::armor`, `inventory::weapon` or `info`. Each item collection can have a different combination of these attributes as the sort key. User `account1234` has an `inventory::weapons` item, while user `account1387` does not (because they have not found any yet). User `account1138` only uses two items for their sort key (since they have no inventory yet) while the other users use three.

DynamoDB lets you selectively retrieve items from these item collections to do the following:

- Retrieve all items from a particular user
- Retrieve only one item from a particular user
- Retrieve all the items of a specific type belonging to a particular user

Speed up queries by organizing your data with item collections

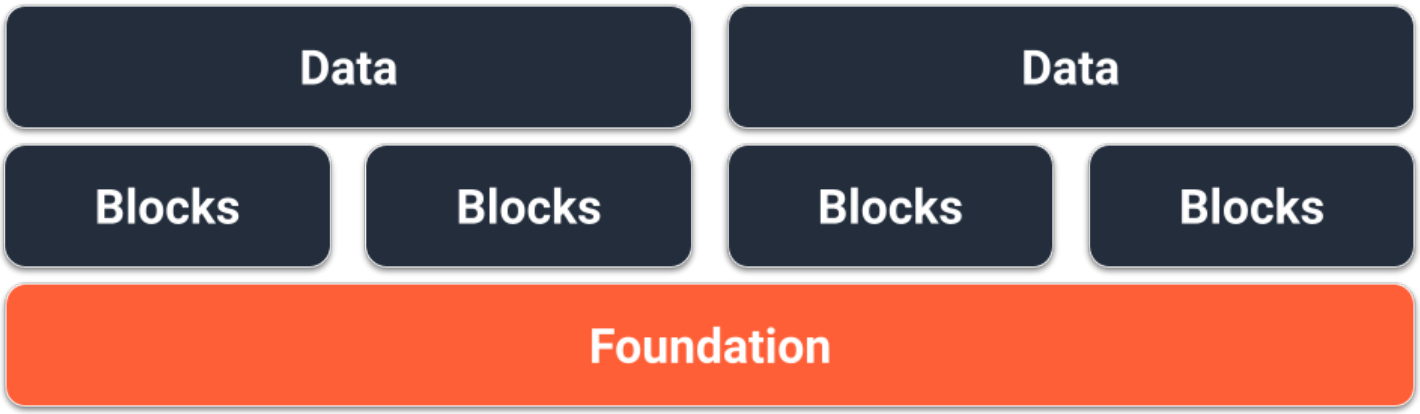
In this example, each of the items in these three item collections represents a player and the data model we have chosen, based off the game's and player's access patterns. What data does the game need? When does it need it? How frequently does it need it? What's the cost of doing it this way? These data modeling decisions were made based off the answers to these questions.

In this game, there is a different page presented to the player for their inventory for weapons and another page for armor. When the player opens their inventory, weapons are shown first because we want that page to load extremely fast, while subsequent inventory pages can load after that. Since each of these item types can be quite large as the player acquires more in-game items, we decided that each inventory page would be its own item in the player's item collection in the database.

The following section talks more about how you can interact with item collections through the `Query` operation.

Data Modeling foundations in DynamoDB

This section covers the foundation layer by examining the two types of table design: single table and multiple table.



Single table design foundation

One choice for the foundation of our DynamoDB schema is **single table design**. Single table design is a pattern that allows you to store multiple types (entities) of data in a single DynamoDB table. It aims to optimize data access patterns, improve performance, and reduce costs by eliminating the need for maintaining multiple tables and complex relationships between them. This is possible because DynamoDB stores items with the same partition key (known as an item collection) on the

same partition(s) as each other. In this design, different types of data are stored as items in the same table, and each item is identified by a unique sort key.

Primary key		Attributes	
Partition key: PK	Sort key: SK		
UserID	Address#USA#CA#LA#90029	data	GSI-SK
		"Street Address"	Default
	Cart#ACTIVE#Coffee	data	GSI-SK
		CoffeeSKU	2019-11-27T103324
	Cart#ACTIVE#Spice	data	GSI-SK
		SpiceSKU	2019-11-28T091245
	Cart#SAVED#Cocoa	data	GSI-SK
		CocoaSKU	2019-11-28T125642
	OrderHistory#OrderUID	data	GSI-SK
		{Order:DataMap}	2019-10-08T132612
	ProfileName	data	
		"Paul Atreides"	
	Store#StoreUID	data	GSI-SK
		Los Angeles	Active

Advantages

- Data locality to support queries for multiple entity types in a single database call
- Reduces overall financial and latency costs of reads:
  - A single query for two items totalling less than 4KB is 0.5 RCU eventually consistent
  - Two queries for two items totalling less than 4KB is 1 RCU eventually consistent (0.5 RCU each)
  - The time to return two separate database calls will average higher than a single call
- Reduces the number of tables to manage:
  - Permissions do not need to be maintained across multiple IAM roles or policies
  - Capacity management for the table is averaged across all entities, usually resulting in a more predictable consumption pattern
  - Monitoring requires fewer alarms
  - Customer Managed Encryption Keys only need to be rotated on one table

- Smooths traffic to the table:
  - By aggregating multiple usage patterns to the same table, the overall usage tends to be smoother (the way a stock index's performance tends to be smoother than any individual stock) which works better for achieving higher utilization with provisioned mode tables

Disadvantages

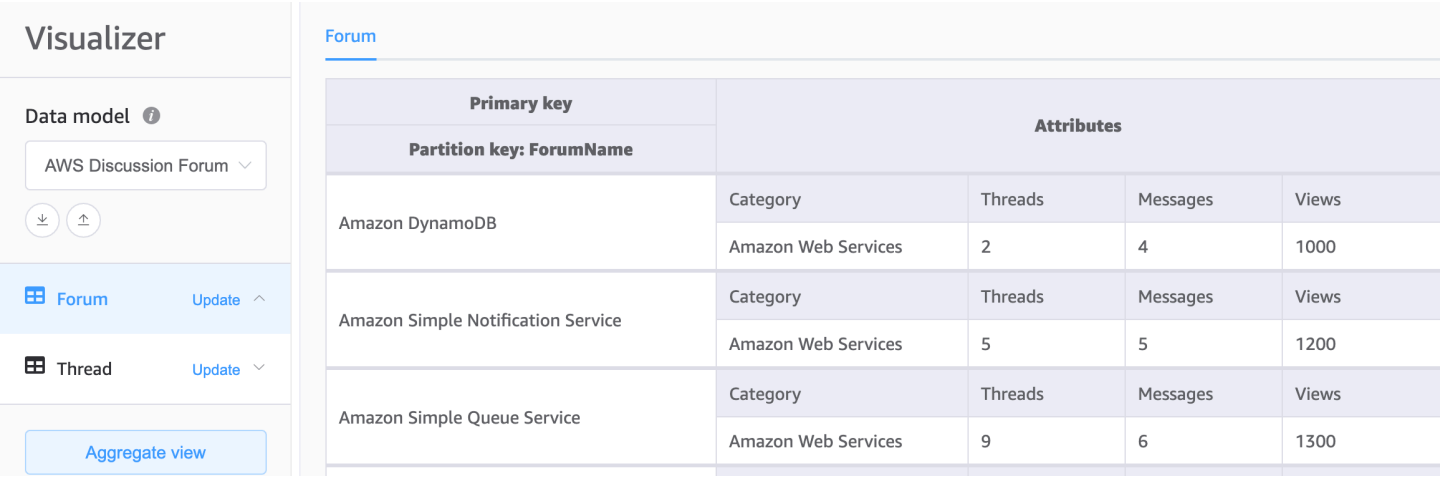
- Learning curve can be steep due to paradoxical design compared to relational databases
- Data requirements must be consistent across all entity types
  - Backups are all or nothing so if some data is not mission critical, consider keeping it in a separate table
  - Table encryption is shared across all items. For multi-tenant applications with individual tenant encryption requirements, client side encryption would be required
  - Tables with a mix of historical data and operational data will not see as much of a benefit from enabling the Infrequent Access Storage Class. For more information, see DynamoDB table classes
- All changed data will be propagated to DynamoDB Streams even if only a subset of entities need to be processed.
  - Thanks to Lambda event filters, this will not affect your bill when using Lambda, but will be an added cost when using the Kinesis Consumer Library
- When using GraphQL, single table design will be more difficult to implement
- When using higher-level SDK clients like Java's `DynamoDBMapper` or Enhanced Client, it can be more difficult to process results because items in the same response may be associated with different classes

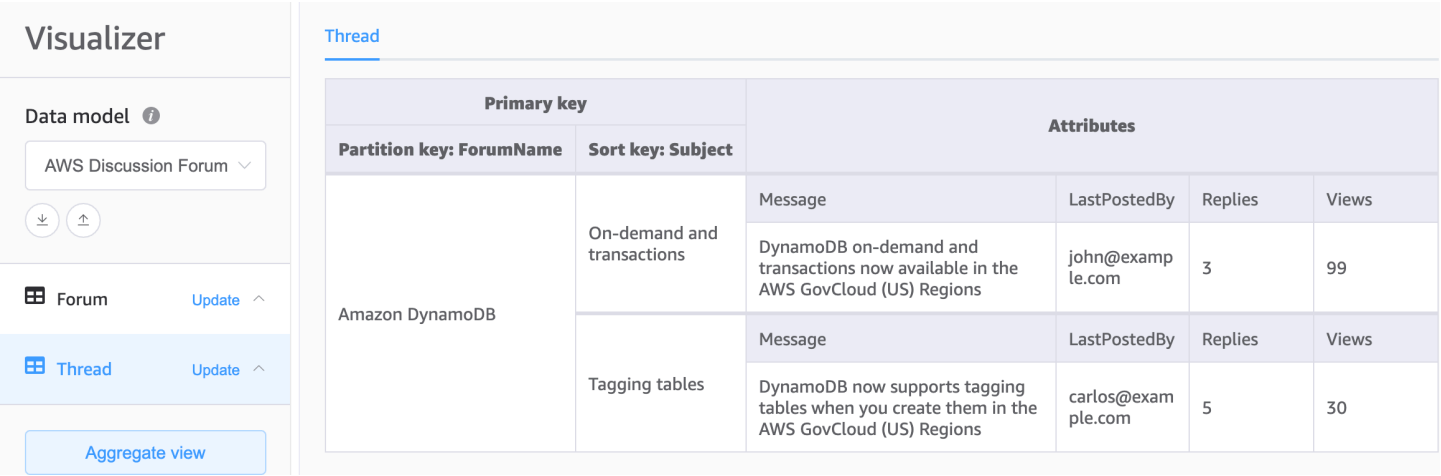
When to use

Single table design works well for applications that frequently query multiple entity types together or need to maintain relationships between different data types. It's particularly effective when your access patterns benefit from data locality and when you want to minimize the overhead of managing multiple tables.

Multiple table design foundation

The second choice for the foundation of our DynamoDB schema is multiple table design. Multiple table design is a pattern that is more like a traditional database design where you store a single type(entity) of data in a each DynamoDB table. Data within each table will still be organized by partition key so performance within a single entity type will be optimized for scalability and performance, but queries across multiple tables must be done independently.





Advantages

- Simpler to design for those who aren't used to working with single table design
- Easier implementation of GraphQL resolvers due to each resolver mapping to a single entity(table)
- Allows for unique data requirements across different entity types:
  - Backups can be made for the individual tables that are mission critical

- Table encryption can be managed for each table. For multi-tenant applications with individual tenant encryption requirements, separate tenant tables make it possible for each customer to have their own encryption key
- Infrequent Access Storage Class can be enabled on just the tables with historical data to realize the full cost savings benefit. For more information, see DynamoDB table classes
- Each table will have its own change data stream allowing for a dedicated Lambda function to be designed for each type of item rather than a single monolithic processor

Disadvantages

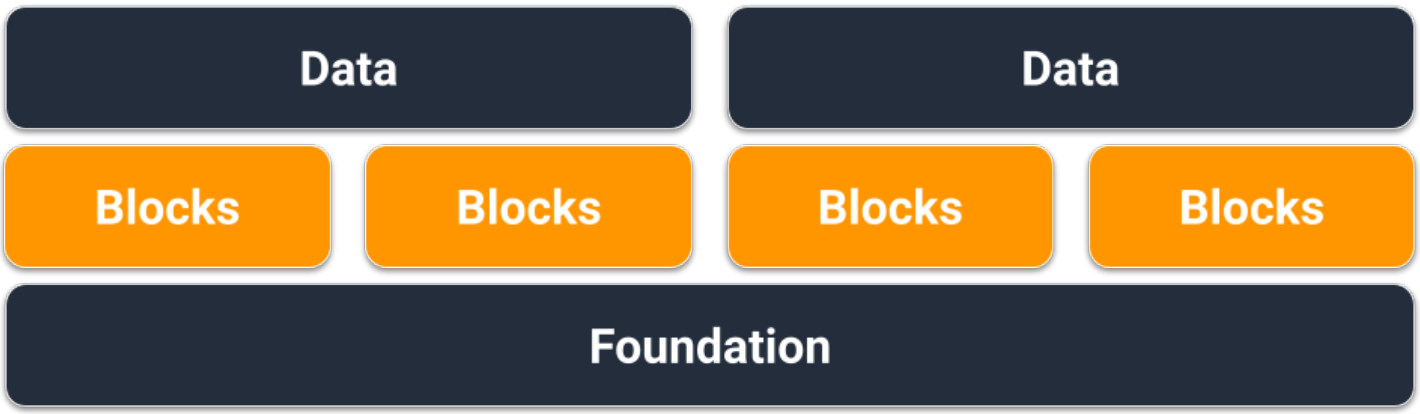
- For access patterns that require data across multiple tables, multiple reads from DynamoDB will be required and data may need to be processed/joined on the client code.
- Operations and monitoring of multiple tables requires more CloudWatch alarms and each table must be scaled independently
- Each tables permissions will need to be managed separately. The addition of tables in the future will require a change to any necessary IAM roles or policies

When to use

If your application's access patterns do not have the need to query multiple entities or tables together, then multiple table design is a good and sufficient approach.

Data modeling building blocks in DynamoDB

This section covers the building block layer to give you design patterns you can use in your application.



Topics

- Composite sort key building block
- Multi-tenancy building block
- Sparse index building block
- Time to live building block
- Time to live for archival building block
- Vertical partitioning building block
- Write sharding building block

Composite sort key building block

When people think of NoSQL, they may also think of it as non-relational. Ultimately, there is no reason relationships cannot be placed into a DynamoDB schema, they just look different than relational databases and their foreign keys. One of the most critical patterns we can use to develop a logical hierarchy of our data in DynamoDB is a composite sort key. The most common style for designing one is with each layer of the hierarchy (parent layer > child layer > grandchild layer) separated by a hashtag. For example, `PARENT#CHILD#GRANDCHILD#ETC`.

Primary key	
Partition key: PK	**Sort key: SK**
UserID	CART#ACTIVE#Apples
	CART#ACTIVE#Bananas
	CART#SAVED#Oranges
	CART#SAVED#Pears
	WISH#VEGGIES#Carrots

While a partition key in DynamoDB always requires the exact value to query for data, we can apply a partial condition to the sort key from left to right similar to traversing a binary tree.

In the example above, we have an e-Commerce store with a Shopping Cart that needs to be maintained across user sessions. Whenever the user logs in, they may want to see the entire

Shopping Cart including items saved for later. But when they enter the checkout, only items in the active cart should be loaded for purchase. Since both of these `KeyConditions` explicitly ask for CART sort keys, the additional wishlist data is simply ignored by DynamoDB at read time. While both saved and active items are a part of the same cart, we need to treat them differently in different parts of the application, so applying a `KeyCondition` to the prefix of the sort key is the most optimized way of retrieving only the data needed for each part of the application.

Key features of this building block

- Related items are stored locally to each other for effective data access
- Using `KeyCondition` expressions, subsets of the hierarchy can be selectively retrieved meaning there are no wasted RCUs
- Different parts of the application can store their items under a specific prefix preventing overwritten items or conflicting writes

Multi-tenancy building block

Many customers use DynamoDB to host data for their multi-tenant applications. For these scenarios, we want to design the schema in a way that keeps all data from a single tenant in its own logical partition of the table. This leverages the concept of an Item Collection, which is a term for all items in a DynamoDB table with the same partition key. For more information on how DynamoDB approaches multitenancy, see Multitenancy on DynamoDB.

Primary key		Attributes
Partition key: PK	**Sort key: SK**	
UserOne	PhotoID1	ImageURL
		https://s3.amazonaws.com/[BUCKET-NAME]/[FILE-NAME].[FILE-TYPE]
	PhotoID2	ImageURL
		https://s3.amazonaws.com/[BUCKET-NAME]/[FILE-NAME].[FILE-TYPE]
UserTwo	PhotoID3	ImageURL
		https://s3.amazonaws.com/[BUCKET-NAME]/[FILE-NAME].[FILE-TYPE]
	PhotoID4	ImageURL
		https://s3.amazonaws.com/[BUCKET-NAME]/[FILE-NAME].[FILE-TYPE]
UserThree	PhotoID5	ImageURL
		https://s3.amazonaws.com/[BUCKET-NAME]/[FILE-NAME].[FILE-TYPE]

For this example, we are running a photo hosting site with potentially thousands of users. Each user will only upload photos to their own profile initially, but by default we will not allow a user to see the photos of any other user. An additional level of isolation would ideally be added to the authorization of each user's call to your API to ensure they are only requesting data from their own partition, but at the schema level, unique partition keys is adequate.

Key features of this building block

- The amount of data read by any one user or tenant can only be as much as the total amount of items in their partition
- Removal of a tenant's data due to an account closure or compliance request can be done tactfully and cheaply. Simply run a query where the partition key equals their tenant ID, then execute a `DeleteItem` operation for each primary key returned

> **Note**
>
> Designed with multi-tenancy in mind, you can use different encryption key providers across a single table to safely isolate data. AWS Database Encryption SDK for Amazon DynamoDB enables you to include client-side encryption in your DynamoDB workloads. You can perform attribute-level encryption, enabling you to encrypt specific attribute values before storing them in your DynamoDB table and search on encrypted attributes without decrypting the entire database beforehand.

Sparse index building block

Sometimes an access pattern requires looking for items that match a rare item or an item that receives a status (which requires an escalated response). Rather than regularly query across the entire dataset for these items, we can leverage the fact that **global secondary indexes (GSI)** are sparsely loaded with data. This means that only items in the base table that have the attributes defined in the index will be replicated to the index.

Primary key		Attributes		
Partition key: DeviceID	Sort key: State#Date			
d#12345	NORMAL#2020-04-24T14:55:00	Operator	Date	
		Liz	2020-04-24	
	WARNING1#2020-04-24T14:45:00	Operator	Date	
		Liz	2020-04-24	
	WARNING1#2020-04-24T14:50:00	Operator	Date	
		Liz	2020-04-24	
d#54321	NORMAL#2020-04-11T06:00:00	Operator	Date	
		Liz	2020-04-11	
	NORMAL#2020-04-11T09:30:00	Operator	Date	
		Sue	2020-04-11	
	WARNING2#2020-04-11T09:25:00	Operator	Date	
		Sue	2020-04-11	
	WARNING3#2020-04-11T05:55:00	Operator	Date	
		Liz	2020-04-11	
d#11223	WARNING4#2020-04-27T16:10:00	Operator	Date	
		Sue	2020-04-27	
	WARNING4#2020-04-27T16:15:00	Operator	Date	EscalatedTo
		Sue	2020-04-27	Sara

Primary key		Attributes	
Partition key: EscalatedTo	Sort key: State#Date		
Sara	WARNING4#2020-04-27T16:15:00	DeviceID	Operator
		d#11223	Sue

In this example, we see an IOT use case where each device in the field is reporting back a status on a regular basis. For the majority of the reports we expect the device to report everything is okay, but on occasion there can be a fault and it must be escalated to a repair technician. For reports with an escalation, the attribute `EscalatedTo` is added the item, but is not present otherwise. The GSI in this example is partitioned by `EscalatedTo` and since the GSI brings over keys from the base table we can still see which DeviceID reported the fault and at what time.

While reads are cheaper than writes in DynamoDB, sparse indexes are a very powerful tool for use cases where instances of a specific type of item is rare but reads to find them are common.

Key features of this building block

- Write and storage costs for the sparse GSI only apply to items that match the key pattern, so the cost of the GSI can be substantially less than other GSIs that have all items replicated to them
- A composite sort key can still be used to further narrow down the items that match the desired query, for instance, a timestamp could be used for the sort key to only view faults reported in the last X minutes (`SK > 5 minutes ago, ScanIndexForward: False`)

Time to live building block

Most data have some duration of time for which it can be considered worth keeping in a primary datastore. To facilitate data aging out from DynamoDB, it has a feature called **time to live (TTL)**. The TTL feature allows you to define a specific attribute at the table level that needs monitoring for items with an epoch timestamp (that's in the past). This allows you to delete expired records from the table for free.

> **Note**
>
> If you are using Global Tables version 2019.11.21 (Current) of global tables and you also use the Time to Live feature, DynamoDB replicates TTL deletes to all replica tables. The initial TTL delete does not consume write capacity in the Region in which the TTL expiry occurs. However, the replicated TTL delete to the replica table(s) consumes replicated write capacity in each of the replica Regions and applicable charges will apply.

Primary key		Attributes	
Partition key: PK	**Sort key: MessageTimestamp**		
UserID	2030-06-30T12:12:12	TTL	Message
		1909570332	Hello
	2030-06-30T12:17:22	TTL	Message
		1909570647	DynamoDB
	2030-06-30T12:22:27	TTL	Message
		1909570947	TTL

In this example, we have an application designed to let a user create messages that are short-lived. When a message is created in DynamoDB, the TTL attribute is set to a date seven days in the future by the application code. In roughly seven days, DynamoDB will see that the epoch timestamp of these items is in the past and delete them.

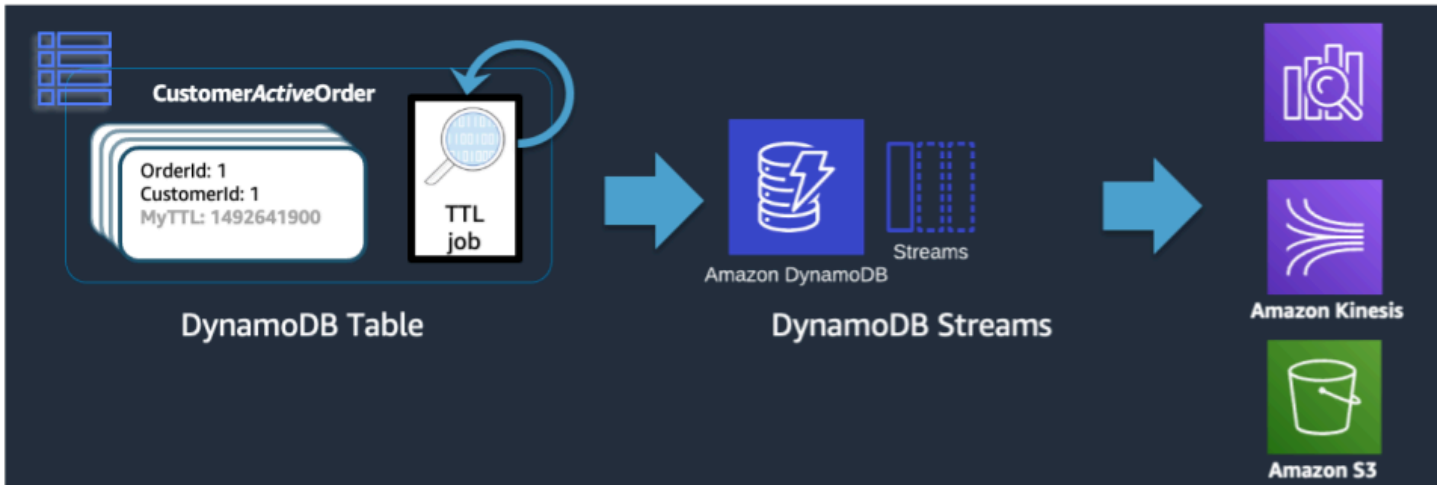
Since the deletes done by TTL are free, it is strongly recommended to use this feature to remove historical data from the table. This will reduce the overall storage bill each month and will likely reduce the costs of user reads since there will be less data to be retrieved by their queries. While TTL is enabled at the table level, it is up to you which items or entities to create a TTL attribute for and how far into the future to set the epoch timestamp to.

Key features of this building block

- TTL deletes are run behind the scenes with no impact to your table performance
- TTL is an asynchronous process that runs roughly every six hours, but can take over 48 hours for an expired record to be deleted
  - Do not rely on TTL deletes for use cases like lock records or state management if stale data must be cleaned up in less than 48 hours
- You can name the TTL attribute a valid attribute name, but the value must be a number type

Time to live for archival building block

While TTL is an effective tool for deleting older data from DynamoDB, many use cases require an archive of the data be kept for a longer period of time than the primary datastore. In this instance, we can leverage TTL's timed deletion of records to push expired records into a long-term datastore.



When a TTL delete is done by DynamoDB, it is still pushed into the DynamoDB Stream as a `Delete` event. When DynamoDB TTL is the one who performs the delete though, there is an attribute on the stream record of `principal:dynamodb`. Using a Lambda subscriber to the DynamoDB Stream, we can apply an event-filter for only the DynamoDB principal attribute and know that any records that match that filter are to be pushed to an archival store like Amazon Glacier.

Key features of this building block

- Once the low-latency reads of DynamoDB are no longer needed for the historical items, migrating them to a colder storage service like Amazon Glacier can reduce storage costs significantly while meeting the data compliance needs of your use case

- If the data is persisted into Amazon S3, cost-efficient analytics tools like Amazon Athena or Redshift Spectrum can be used to perform historical analysis of the data

Vertical partitioning building block

Users familiar with a document model database will be familar with the idea of storing all related data within a single JSON document. While DynamoDB supports JSON data types, it doesn't support executing `KeyConditions` on nested JSON. Since `KeyConditions` are what dictate how much data is read from disk and effectively how many RCUs a query consumes, this can result in inefficiencies at scale. To better optimize the writes and reads of DynamoDB, we recommend breaking apart the document's individual entities into individual DynamoDB items, also referred to as **vertical partitioning**.

```
{
        ”UserProfile” : {
                "FirstName": "Paul",
                "LastName": "Atreides",
                "DateJoined":"1965-08-01"},
        ”Store” : {
                ”store_id”: ”STOREUID”,
                 "city": "Los Angeles",
                 "zip_code":"90029”}
        "ShoppingCart" : [
                {"Spice":
                        { "SKU": "SpiceSKU",
                          "CategoryID":"FictionalSpice",
                          "DateAddded " :"2019-06-11"}},
                {"EspressoBeans":
                        { "SKU": "CaffeineSKU",
                          "CategoryID":"FOODANDDRINK",
                          "DateAddded " :"2019-06-10"}}],
        "ShippingAddress" : {
                 "street_address": "1234 Arrakis Dr",
                 "city": "Los Angeles",
                 "zip_code":"90029",
                 "status": "default"}
        ”OrderHistory#OrderUID” : {
                ”ProductA”: ”SKU_A”,
                ”ProductB”: ”SKU_B”,
                "DateOrdered":”2018-09-28"}
}
```

Primary key		Attributes	
Partition key: PK	Sort key: SK		
UserID	Address#USA#CA#LA#90029	data	GSI-SK
		"Street Address"	Default
	Cart#ACTIVE#Coffee	data	GSI-SK
		CoffeeSKU	2019-11-27T103324
	Cart#ACTIVE#Spice	data	GSI-SK
		SpiceSKU	2019-11-28T091245
	Cart#SAVED#Cocoa	data	GSI-SK
		CocoaSKU	2019-11-28T125642
	OrderHistory#OrderUID	data	GSI-SK
		{Order:DataMap}	2019-10-08T132612
	ProfileName	data	
		"Paul Atreides"	
	Store#StoreUID	data	GSI-SK
		Los Angeles	Active

Vertical partitoning, as shown above, is a key example of single table design in action but can also be implemented across multiple tables if desired. Since DynamoDB bills writes in 1KB increments, you should ideally partition the document in a way that results in items under 1KB.

Key features of this building block

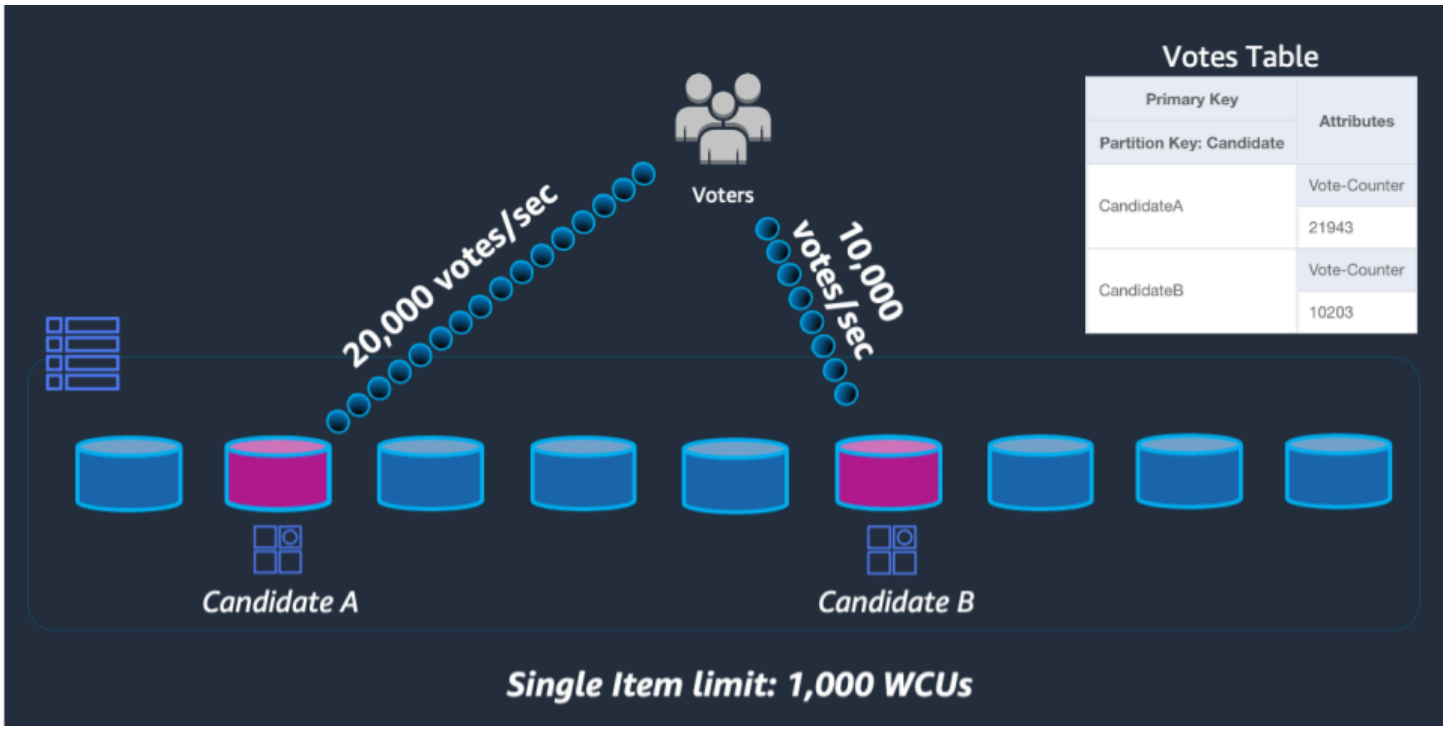
- A hierarchy of data relationships is maintained via sort key prefixes so the singular document structure could be rebuilt client-side if needed
- Singular components of the data structure can be updated independently resulting in small item updates being only 1 WCU
- By using the sort key `BeginsWith`, the application can retrieve similar data in a single query, aggregating read costs for reduced total cost/latency
- Large documents can easily be larger than the 400 KB individual item size limit in DynamoDB and vertical partitioning helps work around this limit

Write sharding building block

One of the very few hard limits DynamoDB has in place is the restriction of how much throughput a single physical partition can maintain per second (not necessarily a single partition key). These limits are presently:

- 1000 WCU (or 1000 <=1KB items written per second) and 3000 RCU (or 3000 <=4KB reads per second) *strongly consistent* or
- 6000 <=4KB reads per second *eventually consistent*

In the event requests against the table exceed either of these limits, an error is sent back to the client SDK of `ThroughputExceededException`, more commonly referred to as throttling. Use cases that require read operations beyond that limit will mostly be served best by placing a read cache in front of DynamoDB, but write operations require a schema level design known as **write sharding**.



Primary Key	Attributes	
Partition Key: Candidate		
CandidateA#1	Vote-Counter	Last-Update
	10238	2019-09-30T11:35:53
CandidateA#2	Vote-Counter	Last-Update
	8452	2019-09-30T11:35:53
CandidateA#3	Vote-Counter	Last-Update
	9148	2019-09-30T11:35:53
CandidateA#4	Vote-Counter	Last-Update
	11092	2019-09-30T11:35:53

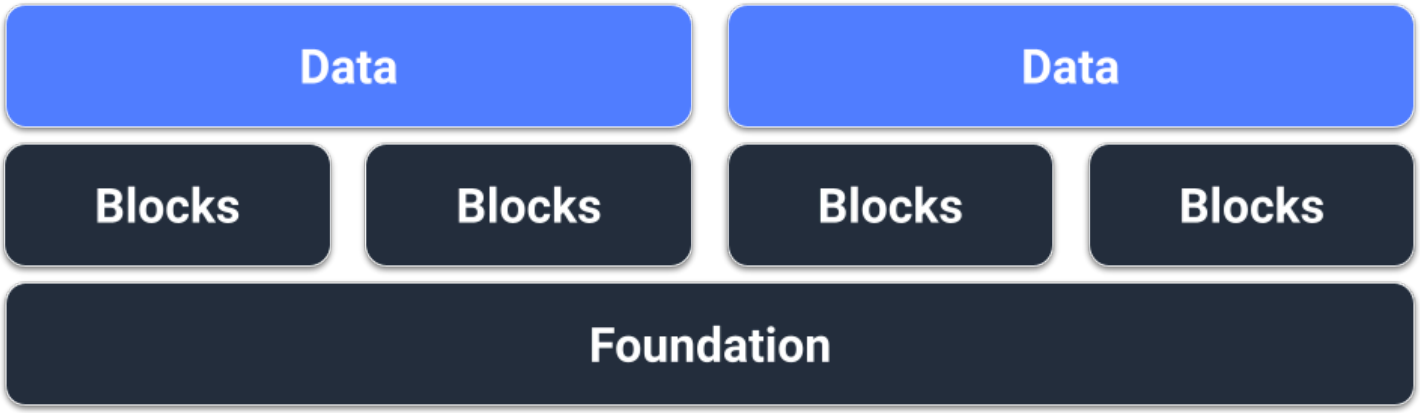
To solve this problem, we'll append a random integer onto the end of the partition key for each contestant in the application's `UpdateItem` code. The range of the random integer generator will need to have an upper bound matching or exceeding the expected amount of writes per second for a given contestant divided by 1000. To support 20,000 votes per second, it would look like rand(0,19). Now that the data is stored under separate logical partitions, it must be combined back together at read time. Since vote totals doesn't need to be real time, a Lambda function scheduled to read all vote partitions every X minutes could perform occasional aggregation for each contestant and write it back to a single vote total record for live reads.

Key features of this building block

- For use cases with extremely high write throughput for a given partition key that cannot be avoided, write operations can be artificially spread across multiple DynamoDB partitions
- GSIs with a low cardinality partition key should also utilize this pattern since throttling on a GSI will apply backpressure to write operations on the base table

Data modeling schema design packages in DynamoDB

Learn about data modeling schema design packages for DynamoDB, including use cases, access patterns, and final schema designs for social networks, gaming profiles, complaint management, recurring payments, device status, and online shops.



Prerequisites

Before we attempt to design our schema for DynamoDB, we must first gather some prerequisite data on the use case the schema needs to support. Unlike relational databases, DynamoDB is sharded by default, meaning that the data will live on multiple servers behind the scenes so designing for data locality is important. We'll need to put together the following list for each schema design:

- List of entities (ER Diagram)
- Estimated volumes and throughput for each entity
- Access patterns that need to be supported (queries and writes)
- Data retention requirements

Topics

- Social network schema design in DynamoDB
- Gaming profile schema design in DynamoDB
- Complaint management system schema design in DynamoDB
- Recurring payments schema design in DynamoDB
- Monitoring device status updates in DynamoDB
- Using DynamoDB as a data store for an online shop

Social network schema design in DynamoDB

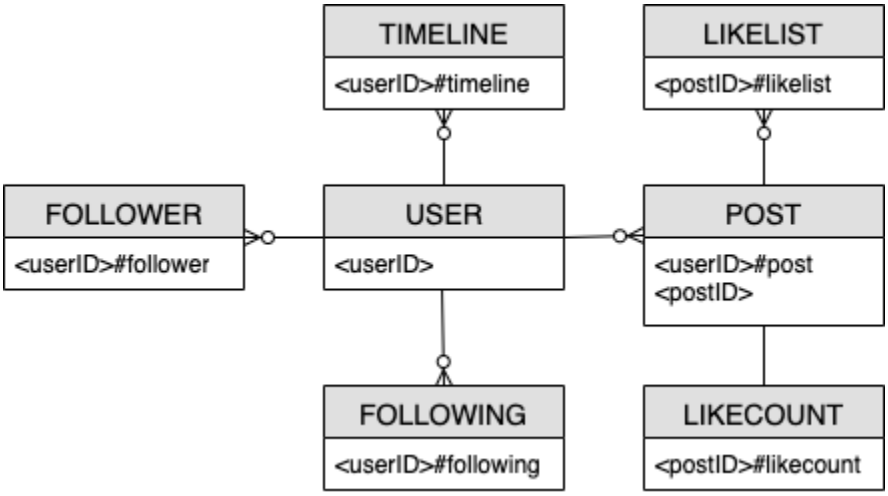
Social network business use case

This use case talks about using DynamoDB as a social network. A social network is an online service that lets different users interact with each other. The social network we'll design will let the user see a timeline consisting of their posts, their followers, who they are following, and the posts written by who they are following. The access patterns for this schema design are:

- Get user information for a given userID
- Get follower list for a given userID
- Get following list for a given userID
- Get post list for a given userID
- Get user list who likes the post for a given postID
- Get the like count for a given postID
- Get the timeline for a given userID

Social network entity relationship diagram

This is the entity relationship diagram (ERD) we'll be using for the social network schema design.



Social network access patterns

These are the access patterns we'll be considering for the social network schema design.

- `getUserInfoByUserID`

- `getFollowerListByUserID`
- `getFollowingListByUserID`
- `getPostListByUserID`
- `getUserLikesByPostID`
- `getLikeCountByPostID`
- `getTimelineByUserID`

Social network schema design evolution

DynamoDB is a NoSQL database, so it does not allow you to perform a join - an operation that combines data from multiple databases. Customers unfamiliar with DynamoDB might apply relational database management system (RDBMS) design philosophies (such as creating a table for each entity) to DynamoDB when they do not need to. The purpose of DynamoDB's single-table design is to write data in a pre-joined form according to the application's access pattern, and then immediately use the data without additional computation. For more information, see Single-table vs. multi-table design in DynamoDB.

Now, let's step through how we'll evolve our schema design to address all the access patterns.

Step 1: Address access pattern 1 (`getUserInfoByUserID`)

To get a given user's information, we'll need to `Query` the base table with a key condition of `PK=<userID>`. The query operation lets you paginate the results, which can be useful when a user has many followers. For more information on Query, see Querying tables in DynamoDB.

In our example, we track two types of data for our user: their "count" and their "info." A user's "count" reflects how many followers they have, how many users they are following, and how many posts they've created. A user's "info" reflects their personal information such as their name.

We see these two kinds of data represented by the two items below. The item that has "count" in its sort key (SK) is more likely to change than the item with "info." DynamoDB considers the size of the item as it appears before and after the update and the provisioned throughput consumed will reflect the larger of these item sizes. So even if you update just a subset of the item's attributes, `UpdateItem` will still consume the full amount of provisioned throughput (the larger of the before and after item sizes). You can get the items via a single `Query` operation and use `UpdateItem` to add or subtract from existing numeric attributes.

Primary key		Attributes			
Partition key: PK	Sort key: SK				
u#12345	"count"	follower#	following#	post#	
		3000000000	971	4945	
	"info"	name	content	imageUrl	etc
		hyuklee	My name is Hyuk Lee	s3://....	...

Step 2: Address access pattern 2 (`getFollowerListByUserID`)

To get a list of users who are following a given user, we'll need to `Query` the base table with a key condition of `PK=<userID>#follower`.

Primary key		Attributes			
Partition key: PK	Sort key: SK				
u#12345	"count"	follower#	following#	post#	
		3000000000	971	4945	
	"info"	name	content	imageUrl	etc
		hyuklee	My name is Hyuk Lee	s3://....	...
u#12345#follower	u#23456				
	u#34567				
	u#45678				

Step 3: Address access pattern 3 (`getFollowingListByUserID`)

To get a list of users a given user is following, we'll need to `Query` the base table with a key condition of `PK=<userID>#following`. You can then use a `TransactWriteItems` operation to group up several requests together and do the following:

- Add User A to User B's follower list, and then increment User B's follower count by one.
- Add User B to User A's follower list, and then increment User A's follower count by one.

Primary key		Attributes			
Partition key: PK	Sort key: SK				
u#12345	"count"	follower#	following#	post#	
		3000000000	971	4945	
	"info"	name	content	imageUrl	etc
		hyuklee	My name is Hyuk Lee	s3://....	...
u#12345#follower	u#23456				
	u#34567				
	u#45678				
u#12345#following	u#56789				
	u#67890				
	u#78912				

Step 4: Address access pattern 4 (`getPostListByUserID`)

To get a list of posts created by a given user, we'll need to `Query` the base table with a key condition of `PK=<userID>#post`. One important thing to note here is that a user's postIDs must be incremental: the second postID value must be greater than the first postID value (since users want to see their posts in a sorted manner). You can do this by generating postIDs based on a time value like a Universally Unique Lexicographically Sortable Identifier (ULID).

Primary key		Attributes			
Partition key: PK	Sort key: SK				
u#12345	"count"	follower#	following#	post#	
		3000000000	971	4945	
	"info"	name	content	imageUrl	etc
		hyuklee	My name is Hyuk Lee	s3://....	...
u#12345#follower	u#23456				
	u#34567				
	u#45678				
u#12345#following	u#56789				
	u#67890				
	u#78912				
u#12345#post	p#12345	content	imageUrl	timestamp	
		content for a post	s3://....	1571827560	
	p#23456	content	imageUrl	timestamp	
		content for a post	s3://....	1571827561	

Step 5: Address access pattern 5 (`getUserLikesByPostID`)

To get a list of users who liked a given user's post, we'll need to `Query` the base table with a key condition of `PK=<postID>#likelist`. This approach is the same pattern that we used for retrieving the follower and following lists in access pattern 2 (`getFollowerListByUserID`) and access pattern 3 (`getFollowingListByUserID`).

Primary key		Attributes			
Partition key: PK	**Sort key: SK**				
u#12345	"count"	follower#	following#	post#	
		3000000000	971	4945	
	"info"	name	content	imageUrl	etc
		hyuklee	My name is Hyuk Lee	s3://....	...
u#12345#follower	u#23456				
	u#34567				
	u#45678				
u#12345#following	u#56789				
	u#67890				
	u#78912				
u#12345#post	p#12345	content	imageUrl	timestamp	
		content for a post	s3://....	1571827560	
	p#23456	content	imageUrl	timestamp	
		content for a post	s3://....	1571827561	
p#12345#likelist	u#23456				
	u#34567				
	u#45678				

Step 6: Address access pattern 6 (`getLikeCountByPostID`)

To get a count of likes for a given post, we'll need to perform a `GetItem` operation on the base table with a key condition of `PK=<postID>#likecount`. This access pattern can cause throttling issues whenever a user with many followers (such as a celebrity) creates a post since throttling occurs when a partition's throughput exceeds 1000 WCU per second. This problem is not a result of DynamoDB, it just appears in DynamoDB since it's at the end of the software stack.

You should evaluate whether it's really essential for all users to view the like count simultaneously or if it can happen gradually over time. In general, a post's like count doesn't need to be immediately 100% accurate. You can implement this strategy by putting a queue between your application and DynamoDB to have the updates happen periodically.

Primary key		Attributes			
Partition key: PK	**Sort key: SK**				
u#12345	"count"	follower#	following#	post#	
		3000000000	971	4945	
	"info"	name	content	imageUrl	etc
		hyuklee	My name is Hyuk Lee	s3://....	...
u#12345#follower	u#23456				
	u#34567				
	u#45678				
u#12345#following	u#56789				
	u#67890				
	u#78912				
u#12345#post	p#12345	content	imageUrl	timestamp	
		content for a post	s3://....	1571827560	
	p#23456	content	imageUrl	timestamp	
		content for a post	s3://....	1571827561	
p#12345#likelist	u#23456				
	u#34567				
	u#45678				
p#12345#likecount	"count"	etc			
		100			

Step 7: Address access pattern 7 (`getTimelineByUserID`)

To get the timeline for a given user, we'll need to perform a `Query` operation on the base table with a key condition of `PK=<userID>#timeline`. Let's consider a scenario where a user's followers need to view their post synchronously. Every time a user writes a post, their follower list is read and their userID and postID are slowly entered into the timeline key of all its followers. Then, when your application starts, you can read the timeline key with the `Query` operation and fill the timeline screen with a combination of userID and postID using the `BatchGetItem` operation for any new items. You cannot read the timeline with an API call, but this is a more cost effective solution if the posts could be edited frequently.

The timeline is a place that shows recent posts, so we'll need a way to clean up the old ones. Instead of using WCU to delete them, you can use DynamoDB's TTL feature to do it for free.

Primary key		Attributes			
Partition key: PK	**Sort key: SK**				
u#12345	"count"	follower#	following#	post#	
		3000000000	971	4945	
	"info"	name	content	imageUrl	etc
		hyuklee	My name is Hyuk Lee	s3://....	...
u#12345#follower	u#23456				
	u#34567				
	u#45678				
u#12345#following	u#56789				
	u#67890				
	u#78912				
u#12345#post	p#12345	content	imageUrl	timestamp	
		content for a post	s3://....	1571827560	
	p#23456	content	imageUrl	timestamp	
		content for a post	s3://....	1571827561	
p#12345#likelist	u#23456				
	u#34567				
	u#45678				
p#12345#likecount	"count"	etc			
		100			
u#12345#timeline	p#34567#u#56789	ttl			
		1571827560			
	p#45678#u#67890	ttl			
		1571827560			
	p#56789#u#78901	ttl			
		1571827560			

All access patterns and how the schema design addresses them are summarized in the table below:

Access pattern	Base table/ GSI/LSI	Operation	Partition key value	Sort key value	Other conditions/ filters
getUserInfoByUserID	Base table	Query	PK=<userID>		

Access pattern	Base table/ GSI/LSI	Operation	Partition key value	Sort key value	Other conditions/ filters
getFollowerListByUserID	Base table	Query	PK=<userID>#follower		
getFollowingListByUserID	Base table	Query	PK=<userID>#following		
getPostListByUserID	Base table	Query	PK=<userID>#post		
getUserLikesByPostID	Base table	Query	PK=<postID>#likelist		
getLikeCountByPostID	Base table	GetItem	PK=<postID>#likecount		
getTimelineByUserID	Base table	Query	PK=<userID>#timeline		

Social network final schema

Here is the final schema design. To download this schema design as a JSON file, see DynamoDB Examples on GitHub.

Base table:

Primary key		Attributes			
Partition key: PK	**Sort key: SK**				
u#12345	"count"	follower#	following#	post#	
		3000000000	971	4945	
	"info"	name	content	imageUrl	etc
		hyuklee	My name is Hyuk Lee	s3://....	...
u#12345#follower	u#23456				
	u#34567				
	u#45678				
u#12345#following	u#56789				
	u#67890				
	u#78912				
u#12345#post	p#12345	content	imageUrl	timestamp	
		content for a post	s3://....	1571827560	
	p#23456	content	imageUrl	timestamp	
		content for a post	s3://....	1571827561	
p#12345#likelist	u#23456				
	u#34567				
	u#45678				
p#12345#likecount	"count"	etc			
		100			
u#12345#timeline	p#34567#u#56789	ttl			
		1571827560			
	p#45678#u#67890	ttl			
		1571827560			
	p#56789#u#78901	ttl			
		1571827560			

Using NoSQL Workbench with this schema design

You can import this final schema into NoSQL Workbench, a visual tool that provides data modeling, data visualization, and query development features for DynamoDB, to further explore and edit your new project. Follow these steps to get started:

1. Download NoSQL Workbench. For more information, see the section called “Download”.
2. Download the JSON schema file listed above, which is already in the NoSQL Workbench model format.
3. Import the JSON schema file into NoSQL Workbench. For more information, see the section called “Importing an existing model”.

4. Once you've imported into NOSQL Workbench, you can edit the data model. For more information, see the section called “Editing an existing model”.

Gaming profile schema design in DynamoDB

Gaming profile business use case

This use case talks about using DynamoDB to store player profiles for a gaming system. Users (in this case, players) need to create profiles before they can interact with many modern games, especially online ones. Gaming profiles typically include the following:

- Basic information such as their user name
- Game data such as items and equipment
- Game records such as tasks and activities
- Social information such as friend lists

To meet the fine-grained data query access requirements for this application, the primary keys (partition key and sort key) will use generic names (PK and SK) so they can be overloaded with various types of values as we will see below.

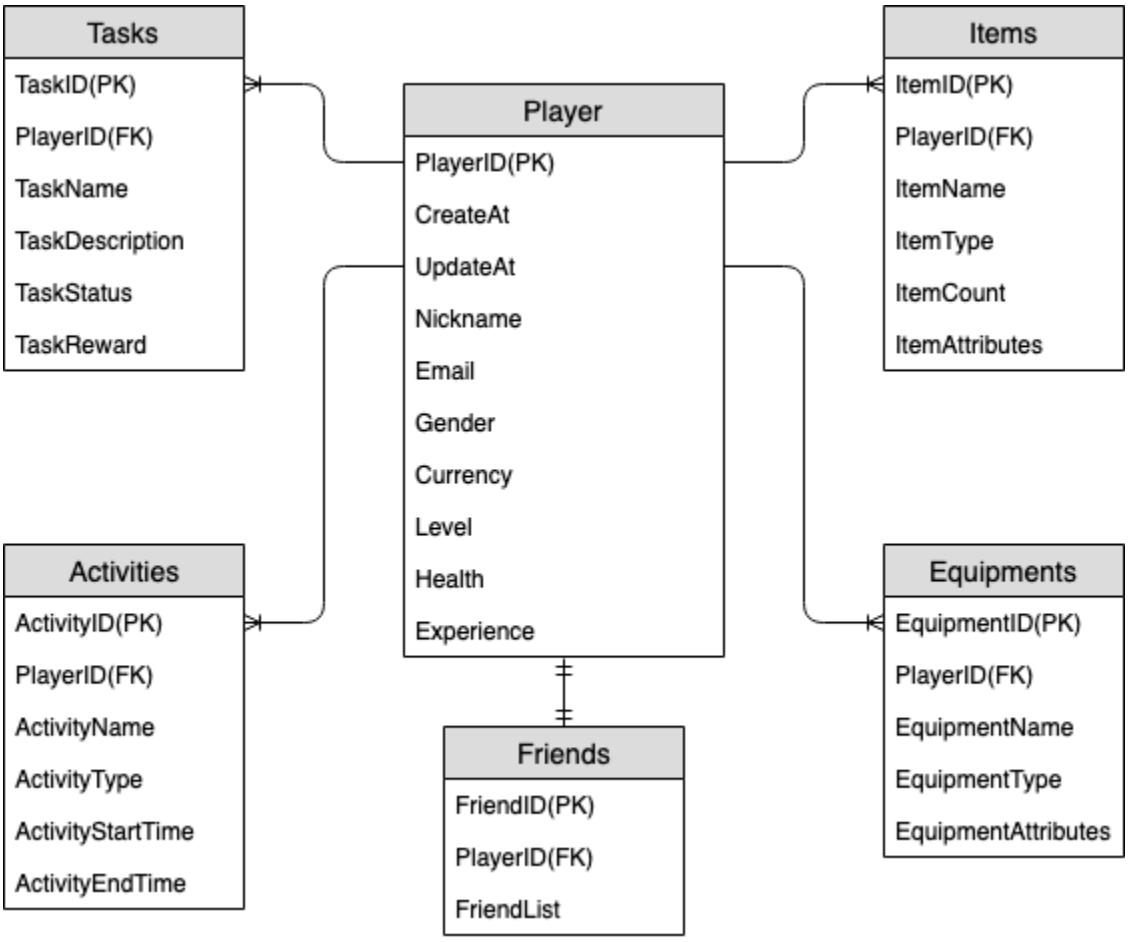
The access patterns for this schema design are:

- Get a user's friend list
- Get all of a player's information
- Get a user's item list
- Get a specific item from the user's item list
- Update a user's character
- Update the item count for a user

The size of the gaming profile will vary in different games. Compressing large attribute values can let them fit within item limits in DynamoDB and reduce costs. The throughput management strategy would depend various on factors such as: number of players, number of games played per second, and seasonality of the workload. Typically for a newly launched game, the number of players and the level of popularity are unknown so we will start with the on-demand throughput mode.

Gaming profile entity relationship diagram

This is the entity relationship diagram (ERD) we'll be using for the gaming profile schema design.



Gaming profile access patterns

These are the access patterns we'll be considering for the social network schema design.

- `getPlayerFriends`
- `getPlayerAllProfile`
- `getPlayerAllItems`
- `getPlayerSpecificItem`
- `updateCharacterAttributes`
- `updateItemCount`

Gaming profile schema design evolution

From the above ERD, we can see that this is a one-to-many relationship type of data modeling. In DynamoDB, one-to-many data models can be organized into item collections, which is different from traditional relational databases where multiple tables are created and linked through foreign keys. An item collections is a group of items that share the same partition key value but have different sort key values. Within an item collection, each item has a unique sort key value that distinguishes it from other items. With this in mind, let's use the following pattern for HASH and RANGE values for each entity type.

To begin, we use generic names like PK and SK to store different types of entities in the same table to make the model future-proof. For better readability, we can include prefixes to denote the type of data or include an arbitrary attribute called `Entity_type` or `Type`. In the current example, we use a string starting with `player` to store `player_ID` as the PK; use `entity name#` as the prefix of SK, and add a `Type` attribute to indicate which entity type this piece of data is. This allows us to support storing more entity types in the future, and use advanced technologies such as GSI Overloading and Sparse GSI to meet more access patterns.

Let's start implementing the access patterns. Access patterns such as adding players and adding equipment can be realized through the `PutItem` operation, so we can ignore them. In this document, we'll focus on the typical access patterns listed above.

Step 1: Address access pattern 1 (`getPlayerFriends`)

We address access pattern 1 (`getPlayerFriends`) with this step. In our current design, friendship is simple and the number of friends in the game is small. For simplicity's sake, we use a list data type to store friend lists (1:1 modeling). In this design, we use `GetItem` to satisfy this access pattern. In the `GetItem` operation, we explicitly provide the partition key and sort key value to get a specific item.

However, if a game has a large number of friends, and the relationships between them are complex (such as friendships being bi-directional with both an invite and accept component) it would be necessary to use a many-to-many relationship to store each friend individually, in order to scale to an unlimited friend list size. And if the friendship change involves operating on multiple items at the same time, DynamoDB transactions can be used to group multiple actions together and submit them as a single all-or-nothing `TransactWriteItems` or `TransactGetItems` operation.

Primary key		Attributes	
Partition key: PK	**Sort key: SK**		
player001	FRIENDS#player001	Type	FriendList
		Friends	[{"M": {"FriendId": {"S":"player002"},"FriendName": {"S":"Alice"}}}, {"M":{"FriendId": {"S":"player003"},"FriendName": {"S":"Bob"}}}]

Step 2: Address access patterns 2 (`getPlayerAllProfile`), 3 (`getPlayerAllItems`), and 4 (`getPlayerSpecificItem`)

We address access patterns 2 (`getPlayerAllProfile`), 3 (`getPlayerAllItems`), and 4 (`getPlayerSpecificItem`) using this step. What these three access patterns have in common is a range query, which uses the `Query` operation. Depending on the scope of the query, Key Condition and Filter Expressions are used, which are commonly used in practical development.

In the Query operation, we provide a single value for Partition Key and get all items with that Partition Key value. Access pattern 2 (`getPlayerAllProfile`) is implemented in this way. Optionally, we can add a sort key condition expression — a string that determines the items to be read from the table. Access pattern 3 (`getPlayerAllItems`) is implemented by adding the key condition of sort key begins_with `ITEMS#`. Further, in order to simplify the development of the application side, we can use filter expressions to implement access pattern 4 (`getPlayerSpecificItem`).

Here's a pseudocode example using filter expression that filters items of the `Weapon` category:

```
filterExpression: "ItemType = :itemType"
expressionAttributeValues: {":itemType": "Weapon"}
```

Primary key		Attributes				
Partition key: PK	**Sort key: SK**					
player001	ITEMS#001	Type	ItemName	ItemType	ItemCount	ItemAttributes
		Item	Health Potion	Consumable	5	{"M":{"HP":{"N":"50"}}}
	ITEMS#002	Type	ItemName	ItemType	ItemCount	ItemAttributes
		Item	Armor of the Knight	Armor	1	{"M":{"DEF":{"N":"100"}}}
	ITEMS#003	Type	ItemName	ItemType	ItemCount	ItemAttributes
		Item	Sword of the Dragon	Weapon	1	{"M":{"ATK":{"N":"100"},"DEF":{"N":"50"}}}

> **Note**
>
> A filter expression is applied after a Query finishes, but before the results are returned to the client. Therefore, a Query consumes the same amount of read capacity regardless of whether a filter expression is present.

If the access pattern is to query a large dataset and filter out a large amount of data to keep only a small subset of data, the appropriate approach is to design DynamoDB Partition Key and Sort Key more effectively. For example, in the above example for obtaining a certain `ItemType`, if there are many items for each player and querying for a certain `ItemType` is a typical access pattern, it would be more efficient to bring `ItemType` into the SK as a composite key. The data model would look like this: `ITEMS#ItemType#ItemId`.

Step 3: Address access patterns 5 (`updateCharacterAttributes`) and 6 (`updateItemCount`)

We address access patterns 5 (`updateCharacterAttributes`) and 6 (`updateItemCount`) using this step. When the player needs to modify the character, such as reducing the currency, or modifying the quantity of a certain weapon in their items, use `UpdateItem` to implement these access patterns. To update a player's currency but ensure it never goes below a minimum amount, we can add a the section called “CLI example” to reduce the balance only if it's greater than or equal to the minimum amount. Here is a pseudocode example:

```
UpdateExpression: "SET currency = currency - :amount"
ConditionExpression: "currency >= :minAmount"
```

Primary key		Attributes									
Partition key: PK	Sort key: SK	Type	CreatedAt	UpdatedAt	Nickname	Email	Gender	Avatar	Currency	PlayerLevel	PlayerHealth	PlayerExperience
player001	#METADATA #player001	Metadata	1618500000	1620000000	John	john@example.com	male	s3://gaming-blki65wn3bgc-lob-avatar/player001.png	500 Updated to 500-Amount	10	80	1000

When developing with DynamoDB and using Atomic Counters to decrement inventory, we can ensure idempotency by using optimistic locking. Here is a pseudocode example for Atomic Counters:

```
UpdateExpression: "SET ItemCount = ItemCount - :incr"
expression-attribute-values: '{":incr":{"N":"1"}}'
```

Primary key		Attributes				
Partition key: PK	Sort key: SK	Type	ItemName	ItemType	ItemCount	ItemAttributes
player001	ITEMS#001	Item	Health Potion	Consumable	5 Updated to 4	{"M":{"HP":{"N":"50"}}}

In addition, in a scenario where the player purchases an item with currency, the entire process needs to deduct currency and add an item at the same time. We can use DynamoDB Transactions to group multiple actions together and submit them as a single all-or-nothing `TransactWriteItems` or `TransactGetItems` operation. `TransactWriteItems` is a synchronous and idempotent write operation that groups up to 100 write actions in a single all-or-nothing operation. The actions are completed atomically so that either all of them succeed or none of them succeeds. Transactions help eliminate the risk of duplication or vanishing currency. For more information on transactions, see DynamoDB transactions example .

All access patterns and how the schema design addresses them are summarized in the table below:

Access pattern	Base table/ GSI/LSI	Operation	Partition key value	Sort key value	Other conditions/ filters
getPlayer Friends	Base table	GetItem	PK=PlayerID	SK="FRIENDS#playerID"	

Access pattern	Base table/ GSI/LSI	Operation	Partition key value	Sort key value	Other conditions/ filters
getPlayer AllProfile	Base table	Query	PK=PlayerID		
getPlayer AllItems	Base table	Query	PK=PlayerID	SK begins_with “ITEMS#”	
getPlayer SpecificItem	Base table	Query	PK=PlayerID	SK begins_with “ITEMS#”	filterExpression: "ItemType = :itemType " expressionAttributeValues: { ":itemType": "Weapon" }
updateCharacterAttributes	Base table	UpdateItem	PK=PlayerID	SK=“#METADATA#playerID”	UpdateExpression: "SET currency = currency - :amount" ConditionExpression: "currency >= :minAmount"

Access pattern	Base table/ GSI/LSI	Operation	Partition key value	Sort key value	Other conditions/ filters
updateItemCount	Base table	UpdateItem	PK=PlayerID	SK =“ITEMS#ItemID”	update-expression: "SET ItemCount = ItemCount - :incr" expression-attribute-values : '{":incr": {"N":"1"}}'

Gaming profile final schema

Here is the final schema design. To download this schema design as a JSON file, see DynamoDB Examples on GitHub.

Base table:

Primary key		Attributes										
Partition key: PK	Sort key: SK											
player001	#METADATA #player001	Type	CreatedAt	UpdatedAt	Nickname	Email	Gender	Avatar	Currency	PlayerLevel	PlayerHealth	PlayerExperience
		Metadata	1618500000	1620000000	John	john@example.com	male	s3://gaming-blki65wn3bgc-iob-avatar/player001.png	500	10	80	1000
	ACTIVITY#001	Type	ActivityEndTime	ActivityName	ActivityReward	ActivityStartTime	ActivityType					
		Activity	1647475199	Hunting Trip	{"M":{"Gold":{"N":"50"},"XP":{"N":"200"}}}	1647388800	Hunting					
	ACTIVITY#002	Type	ActivityEndTime	ActivityName	ActivityReward	ActivityStartTime	ActivityType					
		Activity	1647647999	Mining Adventure	{"M":{"Gold":{"N":"1000"},"XP":{"N":"500"}}}	1647561600	Mining					
	ACTIVITY#003	Type	ActivityEndTime	ActivityName	ActivityReward	ActivityStartTime	ActivityType					
		Activity	1647820799	Arena Challenge	{"M":{"Gold":{"N":"2000"},"XP":{"N":"1000"}}}	1647734400	Arena					
	EQUIPMENTS#001	Type	EquipmentName	EquipmentType	EquipmentAttributes							
		Equipment	Sword of the Dragon	Weapon	{"M":{"ATK":{"N":"100"},"DEF":{"N":"50"}}}							
	EQUIPMENTS#001EQUIPMENTS#002	Type	EquipmentName	EquipmentType	EquipmentAttributes							
		Equipment	Armor of the Knight	Armor	{"M":{"DEF":{"N":"100"}}}							
	EQUIPMENTS#003	Type	EquipmentName	EquipmentType	EquipmentAttributes							
		Equipment	Ring of the Mage	Accessory	{"M":{"SP":{"N":"50"}}}							
	FRIENDS#player001	Type	FriendList									
		Friends	[{"M":{"FriendId":{"S":"player002"},"FriendName":{"S":"Alice"}}},{"M":{"FriendId":{"S":"player003"},"FriendName":{"S":"Bob"}}}]									
	ITEMS#001	Type	ItemName	ItemType	ItemCount	ItemAttributes						
		Item	Health Potion	Consumable	5	{"M":{"HP":{"N":"50"}}}						
	ITEMS#002	Type	ItemName	ItemType	ItemCount	ItemAttributes						
		Item	Armor of the Knight	Armor	1	{"M":{"DEF":{"N":"100"}}}						
	ITEMS#003	Type	ItemName	ItemType	ItemCount	ItemAttributes						
		Item	Sword of the Dragon	Weapon	1	{"M":{"ATK":{"N":"100"},"DEF":{"N":"50"}}}						
	TASK#001	Type	TaskName	TaskDescription	TaskStatus	TaskReward						
		Task	Find the Lost TreasureFind the Lost Treasure	Get clues from a lost adventurer and find the lost treasure.	InProgress	{"M":{"Gold":{"N":"100"},"XP":{"N":"50"}}}						
	TASK#002	Type	TaskName	TaskDescription	TaskStatus	TaskReward						
		Task	Defeat Magic Monsters	Go to the Magic Forest and defeat three magic monsters.	Completed	{"M":{"Gold":{"N":"200"},"XP":{"N":"100"}}}						
	TASK#003	Type	TaskName	TaskDescription	TaskStatus	TaskReward						
		Task	Rescue the Princess	Go to the Demon King's Castle and rescue the princess who is being held captive by the Demon King.	Available	{"M":{"Gold":{"N":"500"},"XP":{"N":"200"}}}						

Using NoSQL Workbench with this schema design

You can import this final schema into NoSQL Workbench, a visual tool that provides data modeling, data visualization, and query development features for DynamoDB, to further explore and edit your new project. Follow these steps to get started:

1. Download NoSQL Workbench. For more information, see the section called “Download”.
2. Download the JSON schema file listed above, which is already in the NoSQL Workbench model format.
3. Import the JSON schema file into NoSQL Workbench. For more information, see the section called “Importing an existing model”.
4. Once you've imported into NOSQL Workbench, you can edit the data model. For more information, see the section called “Editing an existing model”.

Complaint management system schema design in DynamoDB

Complaint management system business use case

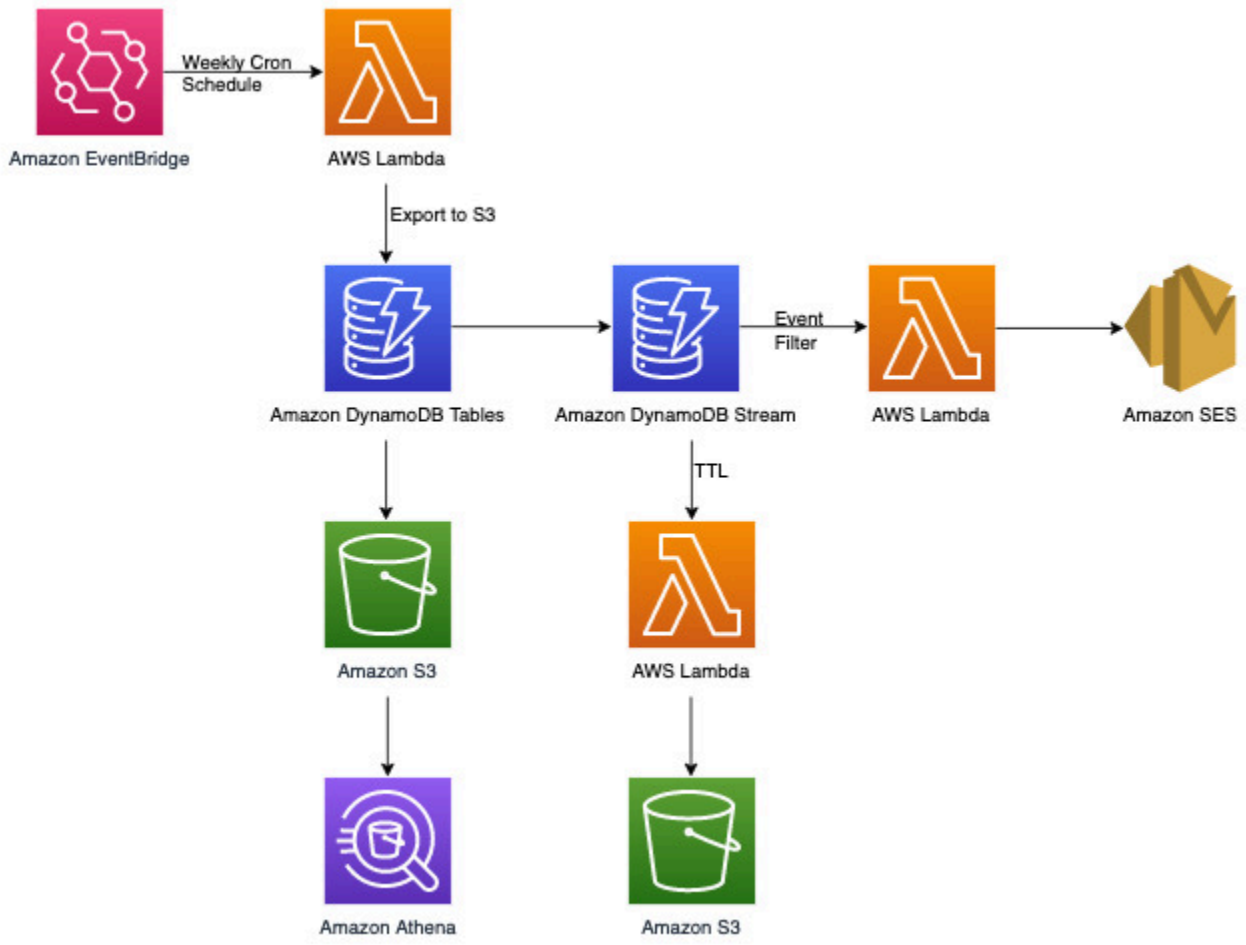
DynamoDB is a database well-suited for a complaint management system (or contact center) use case as most access patterns associated with them would be key-value based transactional lookups. The typical access patterns in this scenario would be to:

- Create and update complaints
- Escalate a complaint
- Create and read comments on a complaint
- Get all complaints by a customer
- Get all comments by an agent and get all escalations

Some comments may have attachments describing the complaint or solution. While these are all key-value access patterns, there can be additional requirements such as sending out notifications when a new comment is added to a complaint or running analytical queries to find complaint distribution by severity (or agent performance) per week. An additional requirement related to lifecycle management or compliance would be to archive complaint data after three years of logging the complaint.

Complaint management system architecture diagram

The following diagram shows the architecture diagram of the complaint management system. This diagram shows the different AWS service integrations that the complaint management system uses.



Apart from the key-value transactional access patterns that we will be handling in the DynamoDB data modeling section later, we have three non-transactional requirements. The architecture diagram above can be broken down into the following three workflows:

1. Send a notification when a new comment is added to a complaint
2. Run analytical queries on weekly data
3. Archive data older than three years

Let's take a more in-depth look at each one.

Send a notification when a new comment is added to a complaint

We can use the below workflow to achieve this requirement:

DynamoDB Streams is a change data capture mechanism to record all write activity on your DynamoDB tables. You can configure Lambda functions to trigger on some or all of these changes. An event filter can be configured on Lambda triggers to filter out events that are not relevant to the use-case. In this instance, we can use a filter to trigger Lambda only when a new comment is added and send out notification to relevant email ID(s) which can be fetched from AWS Secrets Manager or any other credential store.

Run analytical queries on weekly data

DynamoDB is suitable for workloads that are primarily focused on online transactional processing (OLTP). For the other 10-20% access patterns with analytical requirements, data can be exported to S3 with the managed Export to Amazon S3 feature with no impact to the live traffic on DynamoDB table. Take a look at this workflow below:

Amazon EventBridge can be used to trigger AWS Lambda on schedule - it allows you to configure a cron expression for Lambda invocation to take place periodically. Lambda can invoke the `ExportToS3` API call and store DynamoDB data in S3. This S3 data can then be accessed by a SQL

engine such as Amazon Athena to run analytical queries on DynamoDB data without affecting the live transactional workload on the table. A sample Athena query to find number of complaints per severity level would look like this:

```
SELECT Item.severity.S as "Severity", COUNT(Item) as "Count"
FROM "complaint_management"."data"
WHERE NOT Item.severity.S = ''
GROUP BY Item.severity.S ;
```

This results in the following Athena query result:

Results (3)

Search rows

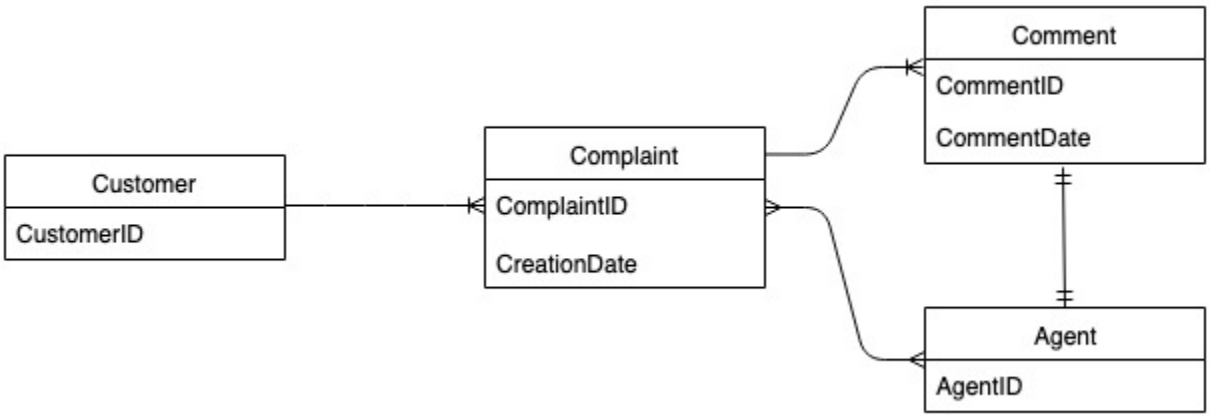
#	Severity	Count
1	P3	1
2	P2	2
3	P1	1

Archive data older than three years

You can leverage the DynamoDB Time to Live (TTL) feature to delete obsolete data from your DynamoDB table at no additional cost (except in the case of global tables replicas for the 2019.11.21 (Current) version, where TTL deletes replicated to other Regions consume write capacity). This data appears and can be consumed from DynamoDB Streams to be archived off into Amazon S3. The workflow for this requirement is as follows:

Complaint management system entity relationship diagram

This is the entity relationship diagram (ERD) we'll be using for the complaint management system schema design.



Complaint management system access patterns

These are the access patterns we'll be considering for the complaint management schema design.

1. createComplaint
2. updateComplaint
3. updateSeveritybyComplaintID
4. getComplaintByComplaintID
5. addCommentByComplaintID
6. getAllCommentsByComplaintID
7. getLatestCommentByComplaintID
8. getAComplaintbyCustomerIDAndComplaintID
9. getAllComplaintsByCustomerID
10. escalateComplaintByComplaintID
11. getAllEscalatedComplaints
12. getEscalatedComplaintsByAgentID (order from newest to oldest)
13. getCommentsByAgentID (between two dates)

Complaint management system schema design evolution

Since this is a complaint management system, most access patterns revolve around a complaint as the primary entity. The `ComplaintID` being highly cardinal will ensure even distribution of data

in the underlying partitions and is also the most common search criteria for our identified access patterns. Therefore, `ComplaintID` is a good partition key candidate in this data set.

Step 1: Address access patterns 1 (`createComplaint`), 2 (`updateComplaint`), 3 (`updateSeveritybyComplaintID`), and 4 (`getComplaintByComplaintID`)

We can use a generic sort key valued called "metadata" (or "AA") to store complaint-specific information such as `CustomerID`, `State`, `Severity`, and `CreationDate`. We use singleton operations with `PK=ComplaintID` and `SK="metadata"` to do the following:

1. `PutItem` to create a new complaint
2. `UpdateItem` to update the severity or other fields in the complaint metadata
3. `GetItem` to fetch metadata for the complaint

Primary key		Attributes				
Partition key: PK	Sort key: SK					
Complaint1321	metadata	customer_id	current_state	creation_time	severity	complaint_description
		custXYZ	assigned	2023-05-10T15:58:00	P2	<Complaint Description>

Step 2: Address access pattern 5 (`addCommentByComplaintID`)

This access pattern requires a one-to-many relationship model between a complaint and comments on the complaint. We will use the vertical partitioning technique here to use a sort key and create an item collection with different types of data. If we look at access patterns 6 (`getAllCommentsByComplaintID`) and 7 (`getLatestCommentByComplaintID`), we know that comments will need to be sorted by time. We can also have multiple comments coming in at the same time so we can use the composite sort key technique to append time and `CommentID` in the sort key attribute.

Other options to deal with such possible comment collisions would be to increase the granularity for the timestamp or add an incremental number as a suffix instead of using `Comment_ID`. In this case, we'll prefix the sort key value for items corresponding to comments with "comm#" to enable range-based operations.

We also need to ensure that the `currentState` in the complaint metadata reflects the state when a new comment is added. Adding a comment might indicate that the complaint has been assigned to an agent or it has been resolved and so on. In order to bundle the addition of comment and

update of current state in the complaint metadata, in an all-or-nothing manner, we will use the TransactWriteItems API. The resulting table state now looks like this:

Primary key		Attributes				
Partition key: PK	**Sort key: SK**					
Complaint1321	comm#2023-05-10T16:00:00#comm3	comm_id	comm_date	complaint_state	comm_text	agentID
		comm3	2023-05-10T16:00:00	investigating	<Comment text>	AgentB
	metadata	customer_id	current_state	creation_time	severity	complaint_description
		custXYZ	investigating	2023-05-10T15:58:00	P2	<Complaint Description>

Let's add some more data in the table and also add `ComplaintID` as a separate field from our PK for future-proofing the model in case we need additional indexes on `ComplaintID`. Also note that some comments may have attachments which we will store in Amazon Simple Storage Service and only maintain their references or URLs in DynamoDB. It's a best practice to keep the transactional database as lean as possible to optimize cost and performance. The data now looks like this:

Primary key		Attributes					
Partition key: PK	**Sort key: SK**						
Complaint123	comm#2023-04-30T12:00:24#comm1	comm_id	comm_date	complaint_state	comm_text	agentID	
		comm1	2023-04-30T12:00:24	investigating	<comm text>	AgentA	
	comm#2023-04-30T12:35:54#comm2	comm_id	comm_date	complaint_state	comm_text	attachments	agentID
		comm2	2023-04-30T12:35:54	resolved	<comm text>	["s3://URL_for_attachment1","s3://URL_for_attachment2"]	AgentA
	metadata	customer_id	complaint_id	current_state	creation_time	severity	complaint_description
		custABC	Complaint123	resolved	2023-04-30T12:00:00	P2	<description text>
Complaint1321	comm#2023-05-10T16:00:00#comm3	comm_id	comm_date	complaint_state	comm_text	agentID	
		comm3	2023-05-10T16:00:00	investigating	<comm text>	AgentB	
	metadata	customer_id	complaint_id	current_state	creation_time	severity	complaint_description
		custXYZ	Complaint1321	investigating	2023-05-10T15:58:00	P2	<descr_text>

Step 3: Address access patterns 6 (`getAllCommentsByComplaintID`) and 7 (`getLatestCommentByComplaintID`)

In order to get all comments for a complaint, we can use the `query` operation with the `begins_with` condition on the sort key. Instead of consuming additional read capacity to read

the metadata entry and then having the overhead of filtering the relevant results, having a sort key condition like this help us only read what we need. For example, a query operation with PK=Complaint123 and SK begins_with comm# would return the following while skipping the metadata entry:

Primary key		Attributes					
Partition key: PK	**Sort key: SK**						
Complaint123	comm#2023-04-30T12:00:24#comm1	comm_id	comm_date	complaint_state	comm_text	agentID	
		comm1	2023-04-30T12:00:24	investigating	<comm text>	AgentA	
	comm#2023-04-30T12:35:54#comm2	comm_id	comm_date	complaint_state	comm_text	attachments	agentID
		comm2	2023-04-30T12:35:54	resolved	<comm text>	["s3://URL_for_attachment1","s3://URL_for_attachment2"]	AgentA
	metadata	customer_id	complaint_id	current_state	creation_time	severity	complaint_description
		custABC	Complaint123	resolved	2023-04-30T12:00:00	P2	<description text>
Complaint1321	comm#2023-05-10T16:00:00#comm3	comm_id	comm_date	complaint_state	comm_text	agentID	
		comm3	2023-05-10T16:00:00	investigating	<comm text>	AgentB	
	metadata	customer_id	complaint_id	current_state	creation_time	severity	complaint_description
		custXYZ	Complaint1321	investigating	2023-05-10T15:58:00	P2	<descr_text>

Since we need the latest comment for a complaint in pattern 7 (`getLatestCommentByComplaintID`), let's use two additional query parameters:

1. `ScanIndexForward` should be set to False to get results sorted in a descending order
2. `Limit` should be set to 1 to get the latest (only one) comment

Similar to access pattern 6 (`getAllCommentsByComplaintID`), we skip the metadata entry using `begins_with comm#` as the sort key condition. Now, you can perform access pattern 7 on this design using the query operation with `PK=Complaint123` and `SK=begins_with comm#`, `ScanIndexForward=False`, and `Limit` 1. The following targeted item will be returned as a result:

Partition key: PK	Sort key: SK	Attributes					
	comm#2023-04-30T12:00:24#comm1	comm_id	comm_date	complaint_state	comm_text	agentID	
		comm1	2023-04-30T12:00:24	investigating	<comm text>	AgentA	
Complaint123	comm#2023-04-30T12:35:54#comm2	comm_id	comm_date	complaint_state	comm_text	attachments	agentID
		comm2	2023-04-30T12:35:54	resolved	<comm text>	["s3://URL_for_attachment1","s3://URL_for_attachment2"]	AgentA
	metadata	customer_id	complaint_id	current_state	creation_time	severity	complaint_description
		custABC	Complaint123	resolved	2023-04-30T12:00:00	P2	<description text>
Complaint1321	comm#2023-05-10T16:00:00#comm3	comm_id	comm_date	complaint_state	comm_text	agentID	
		comm3	2023-05-10T16:00:00	investigating	<comm text>	AgentB	
	metadata	customer_id	complaint_id	current_state	creation_time	severity	complaint_description
		custXYZ	Complaint1321	investigating	2023-05-10T15:58:00	P2	<descr_text>

Let's add more dummy data to the table.

Primary key		Attributes					
Partition key: PK	**Sort key: SK**						
Complaint123	comm#2023-04-30T12:00:24#comm1	comm_id	comm_date	complaint_state	comm_text	agentID	
		comm1	2023-04-30T12:00:24	investigating	<comm text>	AgentA	
	comm#2023-04-30T12:35:54#comm2	comm_id	comm_date	complaint_state	comm_text	attachments	agentID
		comm2	2023-04-30T12:35:54	resolved	<comm text>	["s3://URL_for_attachment1","s3://URL_for_attachment2"]	AgentA
	metadata	customer_id	complaint_id	current_state	creation_time	severity	complaint_description
		custABC	Complaint123	resolved	2023-04-30T12:00:00	P2	<description text>
Complaint1321	comm#2023-05-10T16:00:00#comm3	comm_id	comm_date	complaint_state	comm_text	agentID	
		comm3	2023-05-10T16:00:00	investigating	<comm text>	AgentB	
	metadata	customer_id	complaint_id	current_state	creation_time	severity	complaint_description
		custXYZ	Complaint1321	investigating	2023-05-10T15:58:00	P2	<descr_text>
Complaint1444	comm#2022-12-31T19:32:00#comm4	comm_id	comm_date	complaint_state	comm_text		
		comm4	2022-12-31T19:32:00	waiting	<comm text>		
	comm#2022-12-31T19:40:00#comm5	comm_id	comm_date	complaint_state	comm_text	attachments	agentID
		comm5	2022-12-31T19:40:00	assigned	<comm text>	["s3://URL_for_attachment1"]	AgentC
	metadata	customer_id	complaint_id	current_state	creation_time	severity	complaint_description
		custXY32	Complaint1444	assigned	2022-12-31T19:39:57	P1	<description text>
Complaint0987	metadata	customer_id	complaint_id	current_state	creation_time	severity	complaint_description
		custXYZ	Complaint0987	assigned	2023-06-10T12:30:08	P3	<description text>

Step 4: Address access patterns 8 (`getAComplaintbyCustomerIDAndComplaintID`) and 9 (`getAllComplaintsByCustomerID`)

Access patterns 8 (`getAComplaintbyCustomerIDAndComplaintID`) and 9 (`getAllComplaintsByCustomerID`) introduces a new search criteria: `CustomerID`. Fetching it from the existing table requires an expensive Scan to read all data and then filter relevant items for the `CustomerID` in question. We can make this search more efficient by creating

a global secondary index (GSI) with `CustomerID` as the partition key. Keeping in mind the one-to-many relationship between customer and complaints as well as access pattern 9 (`getAllComplaintsByCustomerID`), `ComplaintID` would be the right candidate for the sort key.

The data in the GSI would look like this:

Primary key		Attributes					
Partition key: customer_id	**Sort key: complaint_id**						
custABC	Complaint123	PK	SK	current_state	creation_time	severity	complaint_description
		Complaint123	metadata	resolved	2023-04-30T12:00:00	P2	<description text>
custXY32	Complaint1444	PK	SK	current_state	creation_time	severity	complaint_description
		Complaint1444	metadata	assigned	2022-12-31T19:39:57	P1	<description text>
custXYZ	Complaint0987	PK	SK	current_state	creation_time	severity	complaint_description
		Complaint0987	metadata	assigned	2023-06-10T12:30:08	P3	<description text>
	Complaint1321	PK	SK	current_state	creation_time	severity	complaint_description
		Complaint1321	metadata	investigating	2023-05-10T15:58:00	P2	<descr_text>

An example query on this GSI for access pattern 8 (`getAComplaintbyCustomerIDAndComplaintID`) would be: `customer_id=custXYZ`, `sort key=Complaint1321`. The result would be:

Primary key		Attributes					
Partition key: customer_id	**Sort key: complaint_id**						
custABC	Complaint123	PK	SK	current_state	creation_time	severity	complaint_description
		Complaint123	metadata	resolved	2023-04-30T12:00:00	P2	<description text>
custXY32	Complaint1444	PK	SK	current_state	creation_time	severity	complaint_description
		Complaint1444	metadata	assigned	2022-12-31T19:39:57	P1	<description text>
	Complaint0987	PK	SK	current_state	creation_time	severity	complaint_description
		Complaint0987	metadata	assigned	2023-06-10T12:30:08	P3	<description text>
custXYZ							
	Complaint1321	PK	SK	current_state	creation_time	severity	complaint_description
		Complaint1321	metadata	investigating	2023-05-10T15:58:00	P2	<descr_text>

To get all complaints for a customer for access pattern 9 (`getAllComplaintsByCustomerID`), the query on the GSI would be: `customer_id=custXYZ` as the partition key condition. The result would be:

Primary key		Attributes					
Partition key: customer_id	**Sort key: complaint_id**						
custABC	Complaint123	PK	SK	current_state	creation_time	severity	complaint_description
		Complaint123	metadata	resolved	2023-04-30T12:00:00	P2	<description text>
custXY32	Complaint1444	PK	SK	current_state	creation_time	severity	complaint_description
		Complaint1444	metadata	assigned	2022-12-31T19:39:57	P1	<description text>
custXYZ	Complaint0987	PK	SK	current_state	creation_time	severity	complaint_description
		Complaint0987	metadata	assigned	2023-06-10T12:30:08	P3	<description text>
	Complaint1321	PK	SK	current_state	creation_time	severity	complaint_description
		Complaint1321	metadata	investigating	2023-05-10T15:58:00	P2	<descr_text>

Step 5: Address access pattern 10 (`escalateComplaintByComplaintID`)

This access introduces the escalation aspect. To escalate a complaint, we can use `UpdateItem` to add attributes such as `escalated_to` and `escalation_time` to the existing complaint metadata item. DynamoDB provides flexible schema design which means a set of non-key attributes can be uniform or discrete across different items. See below for an example:

`UpdateItem with PK=Complaint1444, SK=metadata`

Complaint1444	comm#2022-12-31T19:32:00#comm4	comm_id	comm_date	complaint_state	comm_text			
		comm4	2022-12-31T19:32:00	waiting	<comm text>			
	comm#2022-12-31T19:40:00#comm5	comm_id	comm_date	complaint_state	comm_text	attachments	agentID	
		comm5	2022-12-31T19:40:00	assigned	<comm text>	["s3://URL_for_attachment1"]	AgentC	
	metadata	customer_id	complaint_id	current_state	creation_time	severity	complaint_description	escalated_to	escalation_time
		custXY32	Complaint1444	assigned	2022-12-31T19:39:57	P1	<description text>	AgentB	2023-01-03T04:00:07

Step 6: Address access patterns 11 (`getAllEscalatedComplaints`) and 12 (`getEscalatedComplaintsByAgentID`)

Only a handful of complaints are expected to be escalated out of the whole data set. Therefore, creating an index on the escalation-related attributes would lead to efficient lookups as well as

cost-effective GSI storage. We can do this by leveraging the sparse index technique. The GSI with partition key as `escalated_to` and sort key as `escalation_time` would look like this:

Primary key		Attributes							
Partition key: escalated_to	**Sort key: escalation_time**								
AgentB	2023-01-03T04:00:07	PK	SK	customer_id	complaint_id	current_state	creation_time	severity	complaint_description
		Complaint1444	metadata	custXY32	Complaint1444	assigned	2022-12-31T19:39:57	P1	<description text>
	2023-05-15T14:00:00	PK	SK	customer_id	complaint_id	current_state	creation_time	severity	complaint_description
		Complaint1321	metadata	custXYZ	Complaint1321	investigating	2023-05-10T15:58:00	P2	<descr_text>

To get all escalated complaints for access pattern 11 (`getAllEscalatedComplaints`), we simply scan this GSI. Note that this scan will be performant and cost-efficient due to the size of the GSI. To get escalated complaints for a specific agent (access pattern 12 (`getEscalatedComplaintsByAgentID`)), the partition key would be `escalated_to=agentID` and we set `ScanIndexForward` to `False` for ordering from newest to oldest.

Step 7: Address access pattern 13 (`getCommentsByAgentID`)

For the last access pattern, we need to perform a lookup by a new dimension: `AgentID`. We also need time-based ordering to read comments between two dates so we create a GSI with `agent_id` as the partition key and `comm_date` as the sort key. The data in this GSI will look like the following:

Primary key		Attributes					
Partition key: agentID	**Sort key: comm_date**						
AgentA	2023-04-30T12:00:24	PK	SK	comm_id	complaint_state	comm_text	
		Complaint123	comm#2023-04-30T12:00:24#comm1	comm1	investigating	<comm text>	
	2023-04-30T12:35:54	PK	SK	comm_id	complaint_state	comm_text	attachments
		Complaint123	comm#2023-04-30T12:35:54#comm2	comm2	resolved	<comm text>	["s3://URL_for_attachment1","s3://URL_for_attachment2"]
AgentB	2023-05-10T16:00:00	PK	SK	comm_id	complaint_state	comm_text	
		Complaint1321	comm#2023-05-10T16:00:00#comm3	comm3	investigating	<comm text>	
AgentC	2022-12-31T19:40:00	PK	SK	comm_id	complaint_state	comm_text	attachments
		Complaint1444	comm#2022-12-31T19:40:00#comm5	comm5	assigned	<comm text>	["s3://URL_for_attachment1"]

An example query on this GSI would be `partition key agentID=AgentA and sort key=comm_date between (2023-04-30T12:30:00, 2023-05-01T09:00:00)`, the result of which is:

Primary key		Attributes					
Partition key: agentID	Sort key: comm_date						
AgentA	2023-04-30T12:00:24	PK	SK	comm_id	complaint_state	comm_text	
		Complaint123	comm#2023-04-30T12:00:24#comm1	comm1	investigating	<comm text>	
	2023-04-30T12:35:54	PK	SK	comm_id	complaint_state	comm_text	attachments
		Complaint123	comm#2023-04-30T12:35:54#comm2	comm2	resolved	<comm text>	["s3://URL_for_attachment1","s3://URL_for_attachment2"]
AgentB	2023-05-10T16:00:00	PK	SK	comm_id	complaint_state	comm_text	
		Complaint1321	comm#2023-05-10T16:00:00#comm3	comm3	investigating	<comm text>	
AgentC	2022-12-31T19:40:00	PK	SK	comm_id	complaint_state	comm_text	attachments
		Complaint1444	comm#2022-12-31T19:40:00#comm5	comm5	assigned	<comm text>	["s3://URL_for_attachment1"]

All access patterns and how the schema design addresses them are summarized in the table below:

Access pattern	Base table/ GSI/LSI	Operation	Partition key value	Sort key value	Other conditions/ filters
createComplaint	Base table	PutItem	PK=complaint_id	SK=metadata	
updateComplaint	Base table	UpdateItem	PK=complaint_id	SK=metadata	
updateSeveritybyComplaintID	Base table	UpdateItem	PK=complaint_id	SK=metadata	
getComplaintByComplaintID	Base table	GetItem	PK=complaint_id	SK=metadata	
addCommentByComplaintID	Base table	TransactWriteItems	PK=complaint_id	SK=metadata, SK=comm#comm_date#comm_id	

Access pattern	Base table/ GSI/LSI	Operation	Partition key value	Sort key value	Other conditions/ filters
getAllCommentsByComplaintID	Base table	Query	PK=complaint_id	SK begins_with "comm#"	
getLatestCommentByComplaintID	Base table	Query	PK=complaint_id	SK begins_with "comm#"	scan_index_forward =False, Limit 1
getAComplaintbyCustomerIDAndComplaintID	Customer_complaint_GSI	Query	customer_id=customer_id	complaint_id = complaint_id	
getAllComplaintsByCustomerID	Customer_complaint_GSI	Query	customer_id=customer_id	N/A	
escalateComplaintByComplaintID	Base table	UpdateItem	PK=complaint_id	SK=metadata	
getAllEscalatedComplaints	Escalations_GSI	Scan	N/A	N/A	
getEscalatedComplaintsByAgentID (order from newest to oldest)	Escalations_GSI	Query	escalated_to=agent_id	N/A	scan_index_forward =False

Access pattern	Base table/ GSI/LSI	Operation	Partition key value	Sort key value	Other conditions/ filters
getCommentsByAgentID (between two dates)	Agents_Comments_GSI	Query	agent_id= agent_id	SK between (date1, date2)	

Complaint management system final schema

Here are the final schema designs. To download this schema design as a JSON file, see DynamoDB Examples on GitHub.

Base table

Primary key		Attributes							
Partition key: PK	Sort key: SK								
Complaint123	comm#2023-04-30T12:00:24#comm1	comm_id	comm_date	complaint_state	comm_text	agentID			
		comm1	2023-04-30T12:00:24	investigating	<comm text>	AgentA			
	comm#2023-04-30T12:35:54#comm2	comm_id	comm_date	complaint_state	comm_text	attachments	agentID		
		comm2	2023-04-30T12:35:54	resolved	<comm text>	["s3://URL_for_attachment1","s3://URL_for_attachment2"]	AgentA		
	metadata	customer_id	complaint_id	current_state	creation_time	severity	complaint_description		
		custABC	Complaint123	resolved	2023-04-30T12:00:00	P2	<description text>		
Complaint1321	comm#2023-05-10T16:00:00#comm3	comm_id	comm_date	complaint_state	comm_text	agentID			
		comm3	2023-05-10T16:00:00	investigating	<comm text>	AgentB			
	metadata	customer_id	complaint_id	current_state	creation_time	severity	complaint_description	escalated_to	escalation_time
		custXYZ	Complaint1321	investigating	2023-05-10T15:58:00	P2	<descr_text>	AgentB	2023-05-15T14:00:00
Complaint1444	comm#2022-12-31T19:32:00#comm4	comm_id	comm_date	complaint_state	comm_text				
		comm4	2022-12-31T19:32:00	waiting	<comm text>				
	comm#2022-12-31T19:40:00#comm5	comm_id	comm_date	complaint_state	comm_text	attachments	agentID		
		comm5	2022-12-31T19:40:00	assigned	<comm text>	["s3://URL_for_attachment1"]	AgentC		
	metadata	customer_id	complaint_id	current_state	creation_time	severity	complaint_description	escalated_to	escalation_time
		custXY32	Complaint1444	assigned	2022-12-31T19:39:57	P1	<description text>	AgentB	2023-01-03T04:00:07
Complaint0987	metadata	customer_id	complaint_id	current_state	creation_time	severity	complaint_description		
		custXYZ	Complaint0987	assigned	2023-06-10T12:30:08	P3	<description text>		

Customer_Complaint_GSI

Primary key		Attributes							
Partition key: customer_id	Sort key: complaint_id								
custABC	Complaint123	PK	SK	current_state	creation_time	severity	complaint_description		
		Complaint123	metadata	resolved	2023-04-30T12:00:00	P2	<description text>		
custXY32	Complaint1444	PK	SK	current_state	creation_time	severity	complaint_description	escalated_to	escalation_time
		Complaint1444	metadata	assigned	2022-12-31T19:39:57	P1	<description text>	AgentB	2023-01-03T04:00:07
custXYZ	Complaint0987	PK	SK	current_state	creation_time	severity	complaint_description		
		Complaint0987	metadata	assigned	2023-06-10T12:30:08	P3	<description text>		
	Complaint1321	PK	SK	current_state	creation_time	severity	complaint_description	escalated_to	escalation_time
		Complaint1321	metadata	investigating	2023-05-10T15:58:00	P2	<descr_text>	AgentB	2023-05-15T14:00:00

Escalations_GSI

Primary key		Attributes							
Partition key: escalated_to	Sort key: escalation_time								
AgentB	2023-01-03T04:00:07	PK	SK	customer_id	complaint_id	current_state	creation_time	severity	complaint_description
		Complaint1444	metadata	custXY32	Complaint1444	assigned	2022-12-31T19:39:57	P1	<description text>
	2023-05-15T14:00:00	PK	SK	customer_id	complaint_id	current_state	creation_time	severity	complaint_description
		Complaint1321	metadata	custXYZ	Complaint1321	investigating	2023-05-10T15:58:00	P2	<descr_text>

Agents_Comments_GSI

Primary key		Attributes					
Partition key: agentID	Sort key: comm_date						
AgentA	2023-04-30T12:00:24	PK	SK	comm_id	complaint_state	comm_text	
		Complaint123	comm#2023-04-30T12:00:24#comm1	comm1	investigating	<comm text>	
	2023-04-30T12:35:54	PK	SK	comm_id	complaint_state	comm_text	attachments
		Complaint123	comm#2023-04-30T12:35:54#comm2	comm2	resolved	<comm text>	["s3://URL_for_attachment1","s3://URL_for_attachment2"]
AgentB	2023-05-10T16:00:00	PK	SK	comm_id	complaint_state	comm_text	
		Complaint1321	comm#2023-05-10T16:00:00#comm3	comm3	investigating	<comm text>	
AgentC	2022-12-31T19:40:00	PK	SK	comm_id	complaint_state	comm_text	attachments
		Complaint1444	comm#2022-12-31T19:40:00#comm5	comm5	assigned	<comm text>	["s3://URL_for_attachment1"]

Using NoSQL Workbench with this schema design

You can import this final schema into NoSQL Workbench, a visual tool that provides data modeling, data visualization, and query development features for DynamoDB, to further explore and edit your new project. Follow these steps to get started:

1. Download NoSQL Workbench. For more information, see the section called “Download”.
2. Download the JSON schema file listed above, which is already in the NoSQL Workbench model format.
3. Import the JSON schema file into NoSQL Workbench. For more information, see the section called “Importing an existing model”.
4. Once you've imported into NOSQL Workbench, you can edit the data model. For more information, see the section called “Editing an existing model”.

Recurring payments schema design in DynamoDB

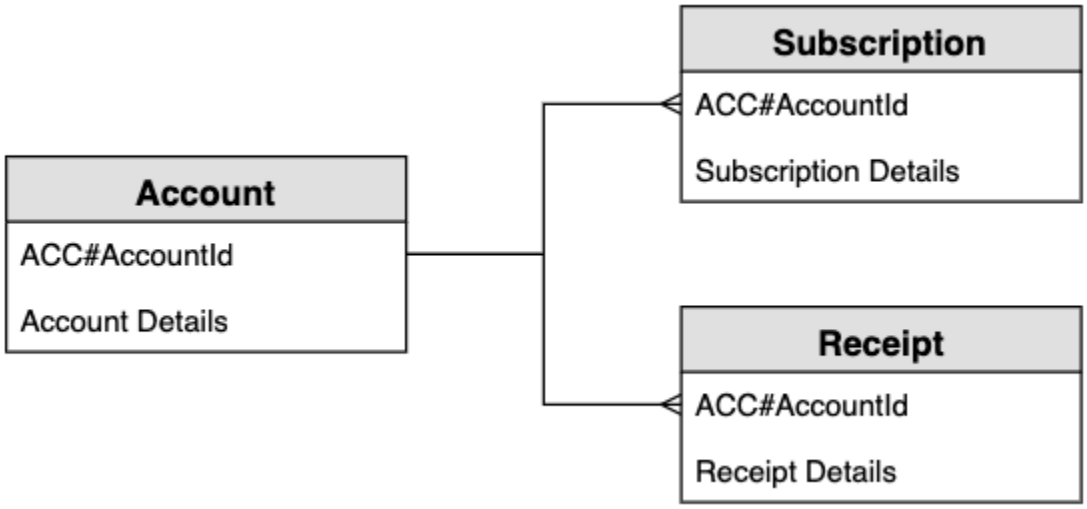
Recurring payments business use case

This use case talks about using DynamoDB to implement a recurring payments system. The data model has the following entities: *accounts*, *subscriptions*, and *receipts*. The specifics for our use case include the following:

- Each *account* can have multiple *subscriptions*
- The *subscription* has a `NextPaymentDate` when the next payment needs to be processed and a `NextReminderDate` when an email reminder is sent to the customer
- There is an item for the *subscription* that is stored and updated when the payment been processed (the average item size is around 1KB and the throughput depends on the number of *accounts* and *subscriptions*)
- The *payment* processor will also create a *receipt* as part of the process which is stored in the table and are set to expire after a period of time by using a TTL attribute

Recurring payments entity relationship diagram

This is the entity relationship diagram (ERD) we'll be using for the recurring payments system schema design.



Recurring payments system access patterns

These are the access patterns we'll be considering for the recurring payments system schema design.

1. `createSubscription`
2. `createReceipt`
3. `updateSubscription`
4. `getDueRemindersByDate`
5. `getDuePaymentsByDate`
6. `getSubscriptionsByAccount`
7. `getReceiptsByAccount`

Recurring payments schema design

The generic names PK and SK are used for key attributes to allow storing different types of entities in the same table such as the account, subscription, and receipt entities. The user first creates a subscription, which is where the user agrees to pay an amount on the same day each month in return for a product. They get the choice on which day of the month to process the payment. There is also a reminder that will be sent prior to the payment being processed. The application works by having two batch jobs that run each day: one batch job sends reminders due that day and the other batch job processes any payments due that day.

Step 1: Address access pattern 1 (`createSubscription`)

Access pattern 1 (`createSubscription`) is used to initially create the subscription, and the details including SKU, `NextPaymentDate`, `NextReminderDate` and `PaymentDetails` are set. This step shows the state of the table for just one account with one subscription. There can be multiple subscriptions in the item collection so this is a one-to-many relationship.

Primary key		Attributes									
Partition key: PK	Sort key: SK	Email	PaymentDay	PaymentAmount	LastPaymentDate	NextPaymentDate	LastReminderDate	NextReminderDate	SKU	PaymentDetails	CreatedDate
ACC#123	SUB#123#SKU#999	s@s.com	28	12.99	1970-01-01T00:00:00.000Z	2023-05-28	1970-01-01T00:00:00.000Z	2023-05-21	999	{"default-card":{"S":"1234123412341234"},"default-address":{"S":"12 Bridge Street, Birmingham, B12 7ST"}}	2023-05-18T09:41:25.856Z

Step 2: Address access patterns 2 (`createReceipt`) and 3 (`updateSubscription`)

Access pattern 2 (`createReceipt`) is used to create the receipt item. After the payment is processed each month, the payment processor will write a receipt back to the base table. There, could be multiple receipts in the item collection so this is a one-to-many relationship. The payment processor will also update the subscription item (access Pattern 3 (`updateSubscription`)) to update for the `NextReminderDate` or the `NextPaymentDate` for the next month.

Primary key		Attributes									
Partition key: PK	Sort key: SK										
ACC#123	REC#12023-05-28T14:15:39.24#SKU#999	Email	SKU	ProcessedDate	ProcessedAmount	TTL					
		s@s.com	999	2023-05-28T14:15:39.247Z	12.99	1700318200					
	SUB#123#SKU#999	Email	PaymentDay	PaymentAmount	LastPaymentDate	NextPaymentDate	LastReminderDate	NextReminderDate	SKU	PaymentDetails	CreatedDate
		s@s.com	28	12.99	2023-05-18T14:15:39.247Z	2023-06-28	2023-05-21T14:15:39.247Z	2023-06-21	999	{"default-card":{"S":"1234123412341234"},"default-address":{"S":"12 Bridge Street, Birmingham, B12 7ST"}}	2023-05-18T09:41:25.856Z

Step 3: Address access pattern 4 (`getDueRemindersByDate`)

The application processes reminders for the payment in batches for the current day. Therefore the application needs to access the subscriptions on a different dimension: date rather than account. This is a good use case for a global secondary index (GSI). In this step we add the index `GSI-1`, which uses the `NextReminderDate` as the GSI partition key. We do not need to replicate all the items. This GSI is a sparse index and the receipts items are not replicated. We also do not need to project all the attributes—we only need to include a subset of the attributes. The image below shows the schema of `GSI-1` and it gives the information needed for the application to send the reminder email.

Primary key		Attributes				
Partition key: NextReminderDate	Sort key: LastReminderDate	SK	PK	SKU	Email	NextPaymentDate
2023-06-21	2023-05-21T14:15:39.247Z	SUB#123#SKU#999	ACC#123	999	s@s.com	2023-06-28

Step 4: Address access pattern 5 (`getDuePaymentsByDate`)

The application processes the payments in batches for the current day in the same way it does with reminders. We add `GSI-2` in this step, and it uses the `NextPaymentDate` as the GSI partition key. We do not need to replicate all the items. This GSI is a sparse index as the receipts items are not replicated. The image below shows the schema of `GSI-2`.

Primary key		Attributes						
Partition key: NextPaymentDate	Sort key: LastPaymentDate	PK	SK	Email	PaymentDay	PaymentAmount	SKU	PaymentDetails
2023-06-28	2023-05-18T14:15:39.247Z	ACC#123	SUB#123#SKU#999	s@s.com	28	12.99	999	{"default-card":{"S":"1234123412341234"},"default-address":{"S":"12 Bridge Street, Birmingham, B12 7ST"}}

Step 5: Address access patterns 6 (`getSubscriptionsByAccount`) and 7 (`getReceiptsByAccount`)

The application can retrieve all the subscriptions for an account by using a query on the base table that targets the account identifier (the PK) and uses the range operator to get all the items where the SK begins with "SUB#". The application can also use the same query structure to retrieve all the receipts by using a range operator to get all the items where the SK begins with "REC#". This allows us to satisfy access patterns 6 (`getSubscriptionsByAccount`) and 7 (`getReceiptsByAccount`). The application uses these access patterns so the user can see their current subscriptions and their past receipts for the last six months. There is no change to the table schema in this step and we can see below how we target just the subscription item(s) in access pattern 6 (`getSubscriptionsByAccount`).

Primary key		Attributes									
Partition key: PK	Sort key: SK	Email	SKU	ProcessedDate	ProcessedAmount	TTL					
	REC#12023-05-28T14:15:39.24#SKU#999	s@s.com	999	2023-05-28T14:15:39.247Z	12.99	1700318200					
ACC#123	SUB#123#SKU#999	Email	PaymentDay	PaymentAmount	LastPaymentDate	NextPaymentDate	LastReminderDate	NextReminderDate	SKU	PaymentDetails	CreatedDate
		s@s.com	28	12.99	2023-05-18T14:15:39.247Z	2023-06-28	2023-05-21T14:15:39.247Z	2023-06-21	999	{"default-card":{"S":"1234123412341234"},"default-address":{"S":"12 Bridge Street, Birmingham, B12 7ST"}}	2023-05-18T09:41:25.856Z

All access patterns and how the schema design addresses them are summarized in the table below:

Access pattern	Base table/GSI/LSI	Operation	Partition key value	Sort key value
createSubscription	Base table	PutItem	ACC#account_id	SUB#<SUBID>#SKU<SKUID>

Access pattern	Base table/GSI/LSI	Operation	Partition key value	Sort key value
createReceipt	Base table	PutItem	ACC#account_id	REC#<ReceiptDate>#SKU<SKUID>
updateSubscription	Base table	UpdateItem	ACC#account_id	SUB#<SUBID>#SKU<SKUID>
getDueRemindersByDate	GSI-1	Query	<NextReminderDate>	
getDuePaymentsByDate	GSI-2	Query	<NextPaymentDate>	
getSubscriptionsByAccount	Base table	Query	ACC#account_id	SK begins_with “SUB#”
getReceiptsByAccount	Base table	Query	ACC#account_id	SK begins_with “REC#”

Recurring payments final schema

Here are the final schema designs. To download this schema design as a JSON file, see DynamoDB Examples on GitHub.

Base table

Primary key		Attributes									
Partition key: PK	Sort key: SK										
ACC#123	REC#12023-05-28T14:15:39.24#SKU#999	Email	SKU	ProcessedDate	ProcessedAmount	TTL					
		s@s.com	999	2023-05-28T14:15:39.247Z	12.99	1700318200					
	SUB#123#SKU#999	Email	PaymentDay	PaymentAmount	LastPaymentDate	NextPaymentDate	LastReminderDate	NextReminderDate	SKU	PaymentDetails	CreatedDate
		s@s.com	28	12.99	2023-05-18T14:15:39.247Z	2023-06-28	2023-05-21T14:15:39.247Z	2023-06-21	999	{"default-card":{"S":"1234123412341234"},"default-address":{"S":"12 Bridge Street, Birmingham, B12 7ST"}}	2023-05-18T09:41:25.856Z

GSI-1

Primary key		Attributes				
Partition key: NextReminderDate	Sort key: LastReminderDate					
2023-06-21	2023-05-21T14:15:39.247Z	SK	PK	SKU	Email	NextPaymentDate
		SUB#123#SKU#999	ACC#123	999	s@s.com	2023-06-28

GSI-2

Primary key		Attributes						
Partition key: NextPaymentDate	Sort key: LastPaymentDate							
2023-06-28	2023-05-18T14:15:39.247Z	PK	SK	Email	PaymentDay	PaymentAmount	SKU	PaymentDetails
		ACC#123	SUB#123#SKU#999	s@s.com	28	12.99	999	{"default-card":{"S":"1234123412341234"},"default-address":{"S":"12 Bridge Street, Birmingham, B12 7ST"}}

Using NoSQL Workbench with this schema design

You can import this final schema into NoSQL Workbench, a visual tool that provides data modeling, data visualization, and query development features for DynamoDB, to further explore and edit your new project. Follow these steps to get started:

1. Download NoSQL Workbench. For more information, see the section called “Download”.
2. Download the JSON schema file listed above, which is already in the NoSQL Workbench model format.
3. Import the JSON schema file into NoSQL Workbench. For more information, see the section called “Importing an existing model”.
4. Once you've imported into NOSQL Workbench, you can edit the data model. For more information, see the section called “Editing an existing model”.

Monitoring device status updates in DynamoDB

This use case talks about using DynamoDB to monitor device status updates (or changes in device state) in DynamoDB.

Use case

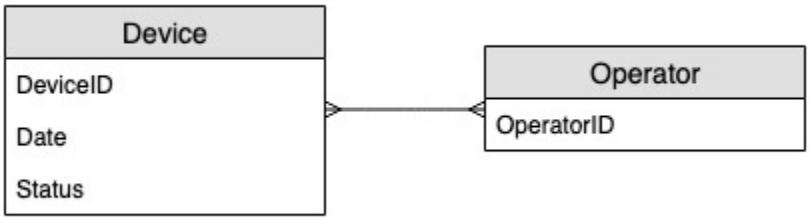
In IoT use-cases (a smart factory for instance) many devices need to be monitored by operators and they periodically send their status or logs to a monitoring system. When there is a problem with a device, the status for the device changes from *normal* to *warning*. There are different log levels or statuses depending on the severity and type of abnormal behavior in the device. The system then assigns an operator to check on the device and they may escalate the problem to their supervisor if needed.

Some typical access patterns for this system include:

- Create log entry for a device
- Get all logs for a specific device state showing the most recent logs first
- Get all logs for a given operator between two dates
- Get all escalated logs for a given supervisor
- Get all escalated logs with a specific device state for a given supervisor
- Get all escalated logs with a specific device state for a given supervisor for a specific date

Entity relationship diagram

This is the entity relationship diagram (ERD) we'll be using for monitoring device status updates.



Access patterns

These are the access patterns we'll be considering for monitoring device status updates.

1. `createLogEntryForSpecificDevice`
2. `getLogsForSpecificDevice`
3. `getWarningLogsForSpecificDevice`
4. `getLogsForOperatorBetweenTwoDates`
5. `getEscalatedLogsForSupervisor`
6. `getEscalatedLogsWithSpecificStatusForSupervisor`
7. `getEscalatedLogsWithSpecificStatusForSupervisorForDate`

Schema design evolution

Step 1: Address access patterns 1 (`createLogEntryForSpecificDevice`) and 2 (`getLogsForSpecificDevice`)

The unit of scaling for a device tracking system would be individual devices. In this system, a `deviceID` uniquely identifies a device. This makes `deviceID` a good candidate for the partition key. Each device sends information to the tracking system periodically (say, every five minutes or so). This ordering makes date a logical sorting criterion and therefore, the sort key. The sample data in this case would look something like this:

Primary key		Attributes
Partition key: DeviceID	Sort key: State#Date	
d#12345	2020-04-24T14:40:00	State
		WARNING1
	2020-04-24T14:45:00	State
		WARNING1
	2020-04-24T14:50:00	State
		WARNING1
	2020-04-24T14:55:00	State
		NORMAL
d#54321	2020-04-11T05:50:00	State
		WARNING3
	2020-04-11T05:55:00	State
		WARNING3
	2020-04-11T06:00:00	State
		NORMAL
	2020-04-11T09:25:00	State
		WARNING2
	2020-04-11T09:30:00	State
		NORMAL
d#11223	2020-04-27T16:10:00	State
		WARNING4
	2020-04-27T16:15:00	State
		WARNING4

To fetch log entries for a specific device, we can perform a query operation with partition key `DeviceID="d#12345"`.

Step 2: Address access pattern 3 (`getWarningLogsForSpecificDevice`)

Since `State` is a non-key attribute, addressing access pattern 3 with the current schema would require a filter expression. In DynamoDB, filter expressions are applied after data is read using

key condition expressions. For example, if we were to fetch warning logs for d#12345, the query operation with partition key `DeviceID="d#12345"` will read four items from the above table and then filter out the one item without the *warning* status. This approach is not efficient at scale. A filter expression can be a good way to exclude items that are queried if the ratio of excluded items is low or the query is performed infrequently. However, for cases where many items are retrieved from a table and the majority of the items are filtered out, we can continue evolving our table design so it runs more efficiently.

Let's change how to handle this access pattern by using composite sort keys. You can import sample data from DeviceStateLog_3.json where the sort key is changed to `State#Date`. This sort key is the composition of the attributes `State`, `#`, and `Date`. In this example, # is used as a delimiter. The data now looks something like this:

Primary key	
Partition key: DeviceID	**Sort key: State#Date**
d#12345	NORMAL#2020-04-24T14:55:00
	WARNING1#2020-04-24T14:40:00
	WARNING1#2020-04-24T14:45:00
	WARNING1#2020-04-24T14:50:00

To fetch only warning logs for a device, the query becomes more targeted with this schema. The key condition for the query uses partition key `DeviceID="d#12345"` and sort key `State#Date begins_with "WARNING"`. This query will only read the relevant three items with the *warning* state.

Step 3: Address access pattern 4 (`getLogsForOperatorBetweenTwoDates`)

You can import DeviceStateLog_4.jsonD where the `Operator` attribute was added to the `DeviceStateLog` table with example data.

Primary key		Attributes		
Partition key: DeviceID	**Sort key: State#Date**			
d#12345	NORMAL#2020-04-24T14:55:00	Operator	Date	State
		Liz	2020-04-24T14:55:00	NORMAL
	WARNING1#2020-04-24T14:40:00	Operator	Date	State
		Liz	2020-04-24T14:40:00	WARNING1
	WARNING1#2020-04-24T14:45:00	Operator	Date	State
		Liz	2020-04-24T14:45:00	WARNING1
	WARNING1#2020-04-24T14:50:00	Operator	Date	State
		Liz	2020-04-24T14:50:00	WARNING1
d#54321	NORMAL#2020-04-11T06:00:00	Operator	Date	State
		Liz	2020-04-11T06:00:00	NORMAL
	NORMAL#2020-04-11T09:30:00	Operator	Date	State
		Sue	2020-04-11T09:30:00	NORMAL
	WARNING2#2020-04-11T09:25:00	Operator	Date	State
		Sue	2020-04-11T09:25:00	WARNING2
	WARNING3#2020-04-11T05:50:00	Operator	Date	State
		Sue	2020-04-11T05:50:00	WARNING3
	WARNING3#2020-04-11T05:55:00	Operator	Date	State
		Liz	2020-04-11T05:55:00	WARNING3
d#11223	WARNING4#2020-04-27T16:10:00	Operator	Date	State
		Sue	2020-04-27T16:10:00	WARNING4
	WARNING4#2020-04-27T16:15:00	Operator	Date	State
		Sue	2020-04-27T16:15:00	WARNING4

Since `Operator` is not currently a partition key, there is no way to perform a direct key-value lookup on this table based on `OperatorID`. We'll need to create a new item collection with a

global secondary index on `OperatorID`. The access pattern requires a lookup based on dates so Date is the sort key attribute for the global secondary index (GSI). This is what the GSI now looks like:

Primary key		Attributes		
Partition key: Operator	**Sort key: Date**			
Liz	2020-04-11T05:55:00	DeviceID	State#Date	State
		d#54321	WARNING3#2020-04-11T05:55:00	WARNING3
	2020-04-11T06:00:00	DeviceID	State#Date	State
		d#54321	NORMAL#2020-04-11T06:00:00	NORMAL
	2020-04-24T14:40:00	DeviceID	State#Date	State
		d#12345	WARNING1#2020-04-24T14:40:00	WARNING1
	2020-04-24T14:45:00	DeviceID	State#Date	State
		d#12345	WARNING1#2020-04-24T14:45:00	WARNING1
	2020-04-24T14:50:00	DeviceID	State#Date	State
		d#12345	WARNING1#2020-04-24T14:50:00	WARNING1
	2020-04-24T14:55:00	DeviceID	State#Date	State
		d#12345	NORMAL#2020-04-24T14:55:00	NORMAL
Sue	2020-04-11T05:50:00	DeviceID	State#Date	State
		d#54321	WARNING3#2020-04-11T05:50:00	WARNING3
	2020-04-11T09:25:00	DeviceID	State#Date	State
		d#54321	WARNING2#2020-04-11T09:25:00	WARNING2
	2020-04-11T09:30:00	DeviceID	State#Date	State
		d#54321	NORMAL#2020-04-11T09:30:00	NORMAL
	2020-04-27T16:10:00	DeviceID	State#Date	State
		d#11223	WARNING4#2020-04-27T16:10:00	WARNING4
	2020-04-27T16:15:00	DeviceID	State#Date	State
		d#11223	WARNING4#2020-04-27T16:15:00	WARNING4

For access pattern 4 (`getLogsForOperatorBetweenTwoDates`), you can query this GSI with partition key `OperatorID=Liz` and sort key `Date` between `2020-04-11T05:58:00` and `2020-04-24T14:50:00`.

Primary key		Attributes		
Partition key: Operator	**Sort key: Date**			
	2020-04-11T05:55:00	DeviceID	State#Date	State
		d#54321	WARNING3#2020-04-11T05:55:00	WARNING3
Liz	2020-04-11T06:00:00	DeviceID	State#Date	State
		d#54321	NORMAL#2020-04-11T06:00:00	NORMAL
	2020-04-24T14:40:00	DeviceID	State#Date	State
		d#12345	WARNING1#2020-04-24T14:40:00	WARNING1
	2020-04-24T14:45:00	DeviceID	State#Date	State
		d#12345	WARNING1#2020-04-24T14:45:00	WARNING1
	2020-04-24T14:50:00	DeviceID	State#Date	State
		d#12345	WARNING1#2020-04-24T14:50:00	WARNING1
	2020-04-24T14:55:00	DeviceID	State#Date	State
		d#12345	NORMAL#2020-04-24T14:55:00	NORMAL
Sue	2020-04-11T05:50:00	DeviceID	State#Date	State
		d#54321	WARNING3#2020-04-11T05:50:00	WARNING3
	2020-04-11T09:25:00	DeviceID	State#Date	State
		d#54321	WARNING2#2020-04-11T09:25:00	WARNING2
	2020-04-11T09:30:00	DeviceID	State#Date	State
		d#54321	NORMAL#2020-04-11T09:30:00	NORMAL
	2020-04-27T16:10:00	DeviceID	State#Date	State
		d#11223	WARNING4#2020-04-27T16:10:00	WARNING4
	2020-04-27T16:15:00	DeviceID	State#Date	State
		d#11223	WARNING4#2020-04-27T16:15:00	WARNING4

Step 4: Address access patterns 5 (`getEscalatedLogsForSupervisor`) 6 (`getEscalatedLogsWithSpecificStatusForSupervisor`), and 7 (`getEscalatedLogsWithSpecificStatusForSupervisorForDate`)

We'll be using a sparse index to address these access patterns.

Global secondary indexes are sparse by default, so only items in the base table that contain primary key attributes of the index will actually appear in the index. This is another way of excluding items that are not relevant for the access pattern being modeled.

You can import DeviceStateLog_6.json where the `EscalatedTo` attribute was added to the `DeviceStateLog` table with example data. As mentioned earlier, not all of the logs gets escalated to a supervisor.

Primary key		Attributes			
Partition key: DeviceID	**Sort key: State#Date**				
d#12345	NORMAL#2020-04-24T14:55:00	Operator	Date	State	
		Liz	2020-04-24T14:55:00	NORMAL	
	WARNING1#2020-04-24T14:40:00	Operator	Date	State	
		Liz	2020-04-24T14:40:00	WARNING1	
	WARNING1#2020-04-24T14:45:00	Operator	Date	State	
		Liz	2020-04-24T14:45:00	WARNING1	
	WARNING1#2020-04-24T14:50:00	Operator	Date	State	
		Liz	2020-04-24T14:50:00	WARNING1	
d#54321	NORMAL#2020-04-11T06:00:00	Operator	Date	State	
		Liz	2020-04-11T06:00:00	NORMAL	
	NORMAL#2020-04-11T09:30:00	Operator	Date	State	
		Sue	2020-04-11T09:30:00	NORMAL	
	WARNING2#2020-04-11T09:25:00	Operator	Date	State	
		Sue	2020-04-11T09:25:00	WARNING2	
	WARNING3#2020-04-11T05:50:00	Operator	Date	State	
		Sue	2020-04-11T05:50:00	WARNING3	
	WARNING3#2020-04-11T05:55:00	Operator	Date	State	
		Liz	2020-04-11T05:55:00	WARNING3	
d#11223	WARNING4#2020-04-27T16:10:00	Operator	Date	State	
		Sue	2020-04-27T16:10:00	WARNING4	
	WARNING4#2020-04-27T16:15:00	Operator	Date	State	EscalatedTo
		Sue	2020-04-27T16:15:00	WARNING4	Sara

You can now create a new GSI where `EscalatedTo` is the partition key and `State#Date` is the sort key. Notice that only items that have both `EscalatedTo` and `State#Date` attributes appear in the index.

Primary key		Attributes			
Partition key: EscalatedTo	Sort key: State#Date	DeviceID	Operator	Date	State
Sara	WARNING4#2020-04-27T16:15:00	d#11223	Sue	2020-04-27T16:15:00	WARNING4

The rest of the access patterns are summarized as follows:

All access patterns and how the schema design addresses them are summarized in the table below:

Access pattern	Base table/ GSI/LSI	Operation	Partition key value	Sort key value	Other conditions/ filters
createLogEntryForSpecificDevice	Base table	PutItem	DeviceID=deviceId	State#Date=state#date	
getLogsForSpecificDevice	Base table	Query	DeviceID=deviceId	State#Date begins_with "state1#"	ScanIndexForward = False
getWarningLogsForSpecificDevice	Base table	Query	DeviceID=deviceId	State#Date begins_with "WARNING"	
getLogsForOperatorBetweenTwoDates	GSI-1	Query	Operator=operatorName	Date between date1 and date2	
getEscalatedLogsForSupervisor	GSI-2	Query	EscalatedTo=supervisorName		

Access pattern	Base table/ GSI/LSI	Operation	Partition key value	Sort key value	Other conditions/ filters
getEscalatedLogsWithSpecificStatusForSupervisor	GSI-2	Query	EscalatedTo=supervisorName	State#Date begins_with "state1#"	
getEscalatedLogsWithSpecificStatusForSupervisorForDate	GSI-2	Query	EscalatedTo=supervisorName	State#Date begins_with "state1#date1"	

Final schema

Here are the final schema designs. To download this schema design as a JSON file, see DynamoDB Examples on GitHub.

Base table

Primary key		Attributes			
Partition key: DeviceID	**Sort key: State#Date**				
d#12345	NORMAL#2020-04-24T14:55:00	Operator	Date	State	
		Liz	2020-04-24T14:55:00	NORMAL	
	WARNING1#2020-04-24T14:40:00	Operator	Date	State	
		Liz	2020-04-24T14:40:00	WARNING1	
	WARNING1#2020-04-24T14:45:00	Operator	Date	State	
		Liz	2020-04-24T14:45:00	WARNING1	
	WARNING1#2020-04-24T14:50:00	Operator	Date	State	
		Liz	2020-04-24T14:50:00	WARNING1	
d#54321	NORMAL#2020-04-11T06:00:00	Operator	Date	State	
		Liz	2020-04-11T06:00:00	NORMAL	
	NORMAL#2020-04-11T09:30:00	Operator	Date	State	
		Sue	2020-04-11T09:30:00	NORMAL	
	WARNING2#2020-04-11T09:25:00	Operator	Date	State	
		Sue	2020-04-11T09:25:00	WARNING2	
	WARNING3#2020-04-11T05:50:00	Operator	Date	State	
		Sue	2020-04-11T05:50:00	WARNING3	
	WARNING3#2020-04-11T05:55:00	Operator	Date	State	
		Liz	2020-04-11T05:55:00	WARNING3	
d#11223	WARNING4#2020-04-27T16:10:00	Operator	Date	State	
		Sue	2020-04-27T16:10:00	WARNING4	
	WARNING4#2020-04-27T16:15:00	Operator	Date	State	EscalatedTo
		Sue	2020-04-27T16:15:00	WARNING4	Sara

GSI-1

Primary key		Attributes		
Partition key: Operator	**Sort key: Date**			
Liz	2020-04-11T05:55:00	DeviceID	State#Date	State
		d#54321	WARNING3#2020-04-11T05:55:00	WARNING3
	2020-04-11T06:00:00	DeviceID	State#Date	State
		d#54321	NORMAL#2020-04-11T06:00:00	NORMAL
	2020-04-24T14:40:00	DeviceID	State#Date	State
		d#12345	WARNING1#2020-04-24T14:40:00	WARNING1
	2020-04-24T14:45:00	DeviceID	State#Date	State
		d#12345	WARNING1#2020-04-24T14:45:00	WARNING1
	2020-04-24T14:50:00	DeviceID	State#Date	State
		d#12345	WARNING1#2020-04-24T14:50:00	WARNING1
	2020-04-24T14:55:00	DeviceID	State#Date	State
		d#12345	NORMAL#2020-04-24T14:55:00	NORMAL
Sue	2020-04-11T05:50:00	DeviceID	State#Date	State
		d#54321	WARNING3#2020-04-11T05:50:00	WARNING3
	2020-04-11T09:25:00	DeviceID	State#Date	State
		d#54321	WARNING2#2020-04-11T09:25:00	WARNING2
	2020-04-11T09:30:00	DeviceID	State#Date	State
		d#54321	NORMAL#2020-04-11T09:30:00	NORMAL
	2020-04-27T16:10:00	DeviceID	State#Date	State
		d#11223	WARNING4#2020-04-27T16:10:00	WARNING4
	2020-04-27T16:15:00	DeviceID	State#Date	State
		d#11223	WARNING4#2020-04-27T16:15:00	WARNING4

GSI-2

Primary key		Attributes			
Partition key: EscalatedTo	Sort key: State#Date	DeviceID	Operator	Date	State
Sara	WARNING4#2020-04-27T16:15:00	d#11223	Sue	2020-04-27T16:15:00	WARNING4

Using NoSQL Workbench with this schema design

You can import this final schema into NoSQL Workbench, a visual tool that provides data modeling, data visualization, and query development features for DynamoDB, to further explore and edit your new project. Follow these steps to get started:

1. Download NoSQL Workbench. For more information, see the section called “Download”.
2. Download the JSON schema file listed above, which is already in the NoSQL Workbench model format.
3. Import the JSON schema file into NoSQL Workbench. For more information, see the section called “Importing an existing model”.
4. Once you've imported into NOSQL Workbench, you can edit the data model. For more information, see the section called “Editing an existing model”.

Using DynamoDB as a data store for an online shop

This use case talks about using DynamoDB as a data store for an online shop (or e-store).

Use case

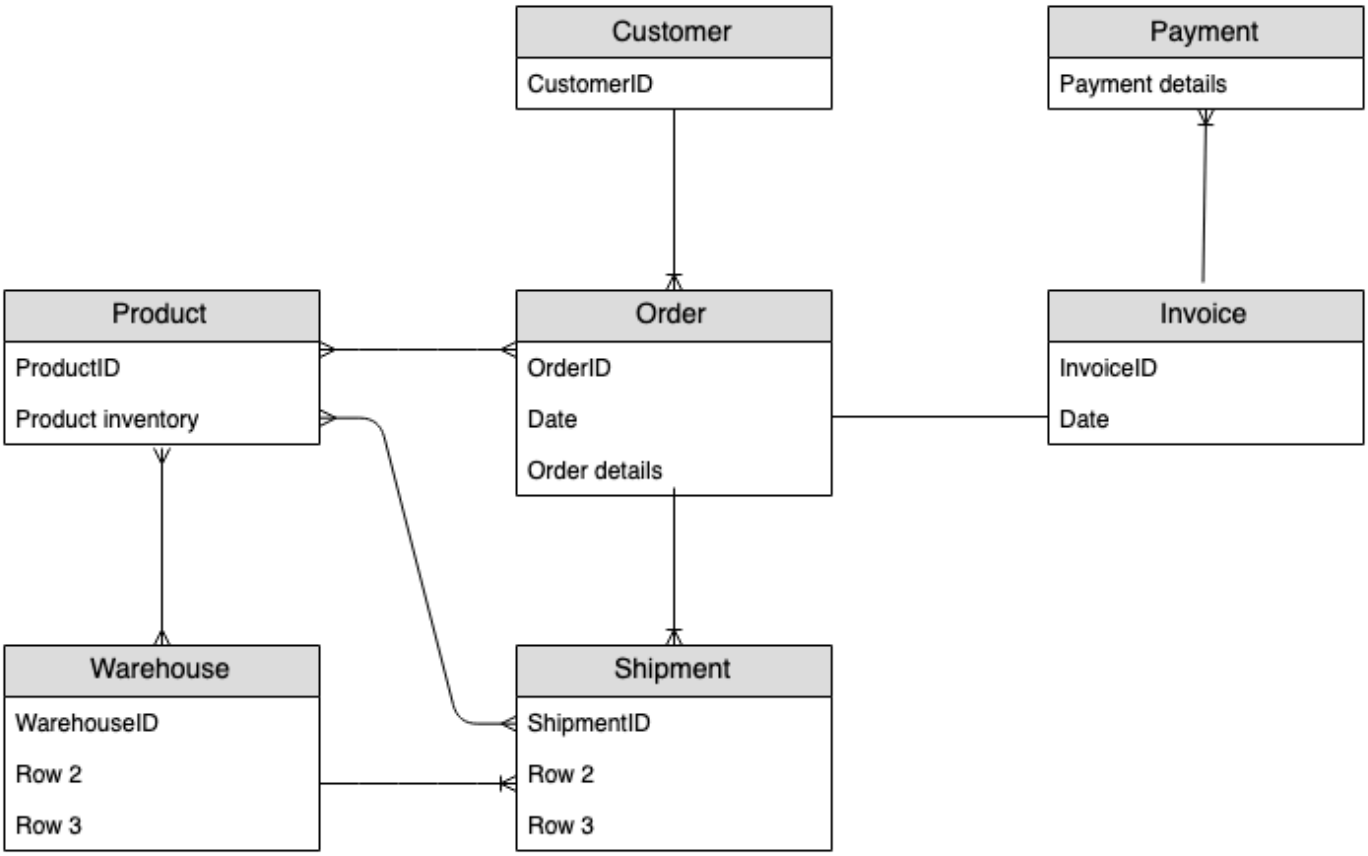
An online store lets users browse through different products and eventually purchase them. Based on the generated invoice, a customer can pay using a discount code or gift card and then pay the remaining amount with a credit card. Purchased products will be picked from one of several warehouses and will be shipped to the provided address. Typical access patterns for an online store include:

- Get customer for a given customerId
- Get product for a given productId
- Get warehouse for a given warehouseId

- Get a product inventory for all warehouses by a productId
- Get order for a given orderId
- Get all products for a given orderId
- Get invoice for a given orderId
- Get all shipments for a given orderId
- Get all orders for a given productId for a given date range
- Get invoice for a given invoiceId
- Get all payments for a given invoiceId
- Get shipment details for a given shipmentId
- Get all shipments for a given warehouseId
- Get inventory of all products for a given warehouseId
- Get all invoices for a given customerId for a given date range
- Get all products ordered by a given customerId for a given date range

Entity relationship diagram

This is the entity relationship diagram (ERD) we'll be using to model DynamoDB as a data store for an online shop.



Access patterns

These are the access patterns we'll be considering when using DynamoDB as a data store for an online shop.

1. `getCustomerByCustomerId`
2. `getProductByProductId`
3. `getWarehouseByWarehouseId`
4. `getProductInventoryByProductId`
5. `getOrderDetailsByOrderId`
6. `getProductByOrderId`
7. `getInvoiceByOrderId`
8. `getShipmentByOrderId`
9. `getOrderByProductIdForDateRange`
10. `getInvoiceByInvoiceId`

11getPaymentByInvoiceId

12getShipmentDetailsByShipmentId

13getShipmentByWarehouseId

14getProductInventoryByWarehouseId

15getInvoiceByCustomerIdForDateRange

16getProductsByCustomerIdForDateRange

Schema design evolution

Using NoSQL Workbench for DynamoDB , import AnOnlineShop_1.json to create a new data model called AnOnlineShop and a new table called OnlineShop. Note that we use the generic names PK and SK for the partition key and sort key. This is a practice used in order to store different types of entities in the same table.

Step 1: Address access pattern 1 (getCustomerByCustomerId)

Import AnOnlineShop_2.json to handle access pattern 1 (getCustomerByCustomerId). Some entities do not have relationships to other entities, so we will use the same value of PK and SK for them. In the example data, note that the keys use a prefix c# in order to distinguish the customerId from other entities that will be added later. This practice is repeated for other entities as well.

To address this access pattern, a GetItem operation can be used with PK=customerId and SK=customerId.

Step 2: Address access pattern 2 (getProductByProductId)

Import AnOnlineShop_3.json to address access pattern 2 (getProductByProductId) for the product entity. The product entities are prefixed by p# and the same sort key attribute has been used to store customerID as well as productID. Generic naming and vertical partitioning allows us to create such item collections for an effective single table design.

To address this access pattern, a GetItem operation can be used with PK=productId and SK=productId.

Step 3: Address access pattern 3 (getWarehouseByWarehouseId)

Import AnOnlineShop_4.json to address access pattern 3 (getWarehouseByWarehouseId) for the warehouse entity. We currently have the customer, product, and warehouse entities added

to the same table. They are distinguished using prefixes and the `EntityType` attribute. A type attribute (or prefix naming) improves the model's readability. The readability would be affected if we simply stored alphanumeric IDs for different entities in the same attribute. It would be difficult to tell one entity from the other in the absence of these identifiers.

To address this access pattern, a `GetItem` operation can be used with `PK=warehouseId` and `SK=warehouseId`.

Base table:

Primary key		Attributes		
Partition key: PK	**Sort key: SK**			
c#12345	c#12345	EntityType	Email	Name
		customer	samaneh@example.com	Samaneh Utter
p#12345	p#12345	EntityType	Detail	Price
		product	{"Name":{"S":"Options Open"},"Description":{"S":"The latest album"}}	100
w#12345	w#12345	EntityType	Address	
		warehouse	{"Country":{"S":"Sweden"},"County":{"S":"Vastra Gotaland"},"City":{"S":"Goteborg"},"Street":{"S":"MainStreet"},"Number":{"S":"20"},"ZipCode":{"S":"41111"}}	

Step 4: Address access pattern 4 (`getProductInventoryByProductId`)

Import AnOnlineShop_5.json to address access pattern 4 (`getProductInventoryByProductId`). `warehouseItem` entity is used to keep track of the number of products in each warehouse. This item would normally be updated when a product is added or removed from a warehouse. As seen in the ERD, there is a many-to-many relationship between `product` and `warehouse`. Here, the one-to-many relationship from `product` to `warehouse` is modeled as `warehouseItem`. Later on, the one-to-many relationship from `warehouse` to `product` will be modeled as well.

Access pattern 4 can be addressed with a query on `PK=ProductId` and `SK begins_with "w#"`.

For more information about `begins_with()` and other expressions that can be applied to sort keys, see Key Condition Expressions.

Base table:

Primary key		Attributes		
Partition key: PK	Sort key: SK			
c#12345	c#12345	EntityType	Email	Name
		customer	samaneh@example.com	Samaneh Utter
	p#12345	EntityType	Detail	Price
		product	{"Name":{"S":"Options Open"},"Description":{"S":"The latest album"}}	100
p#12345	w#12345	EntityType	Quantity	
		warehouseItem	50	
w#12345	w#12345	EntityType	Address	
		warehouse	{"Country":{"S":"Sweden"},"County":{"S":"Vastra Gotaland"},"City":{"S":"Goteborg"},"Street":{"S":"MainStreet"},"Number":{"S":"20"},"ZipCode":{"S":"41111"}}	

Step 5: Address access patterns 5 (`getOrderDetailsByOrderId`) and 6 (`getProductByOrderId`)

Add some more `customer`, `product`, and `warehouse` items to the table by importing AnOnlineShop_6.json. Then, import AnOnlineShop_7.json to build an item collection for `order` that can address access patterns 5 (`getOrderDetailsByOrderId`) and 6 (`getProductByOrderId`). You can see the one-to-many relationship between `order` and `product` modeled as orderItem entities.

To address access pattern 5 (`getOrderDetailsByOrderId`), query the table with `PK=orderId`. This will provide all information about the order including `customerId` and ordered products.

Base table:

Primary key		Attributes		
Partition key: PK	Sort key: SK			
o#12345	c#12345	EntityType	Date	
		order	2020-06-21T19:10:00	
	p#12345	EntityType	Price	Quantity
		orderItem	100	2
	p#99887	EntityType	Price	Quantity
		orderItem	40	5

To address access pattern 6 (`getProductByOrderId`), we need to read products in an `order` only. Query the table with `PK=orderId` and `SK begins_with "p#"` to accomplish this.

Base table:

Primary key		Attributes		
Partition key: PK	Sort key: SK			
	c#12345	EntityType	Date	
		order	2020-06-21T19:10:00	
o#12345	p#12345	EntityType	Price	Quantity
		orderItem	100	2
	p#99887	EntityType	Price	Quantity
		orderItem	40	5

Step 6: Address access pattern 7 (`getInvoiceByOrderId`)

Import AnOnlineShop_8.json to add an `invoice` entity to the *order* item collection to handle access pattern 7 (`getInvoiceByOrderId`). To address this access pattern, you can use a query operation with `PK=orderId` and `SK begins_with "i#"`.

Base table:

Primary key		Attributes		
Partition key: PK	Sort key: SK			
o#12345	c#12345	EntityType	Date	
		order	2020-06-21T19:10:00	
	i#55443	EntityType	Amount	Date
		invoice	400	2020-06-21T19:18:00
	p#12345	EntityType	Price	Quantity
		orderItem	100	2
	p#99887	EntityType	Price	Quantity
		orderItem	40	5

Step 7: Address access pattern 8 (`getShipmentByOrderId`)

Import AnOnlineShop_9.json to add `shipment` entities to the *order* item collection to address access pattern 8 (`getShipmentByOrderId`). We are extending the same vertically partitioned model by adding more types of entities in the single table design. Notice how the *order* item collection contains the different relationships that an `order` entity has with the `shipment`, `orderItem`, and `invoice` entities.

To get shipments by `orderId`, you can perform a query operation with `PK=orderId` and SK `begins_with “sh#”`.

Base table:

Primary key		Attributes				
Partition key: PK	Sort key: SK					
o#12345	c#12345	EntityType	Date			
		order	2020-06-21T19:10:00			
	i#55443	EntityType	Amount	Date		
		invoice	400	2020-06-21T19:18:00		
	p#12345	EntityType	Price	Quantity		
		orderItem	100	2		
	p#99887	EntityType	Price	Quantity		
		orderItem	40	5		
	sh#88899	EntityType	Address	Type	Date	Warehouseld
		shipment	{"Country":{"S":"Sweden"},"County":{"S":"Vastra Gotaland"},"City":{"S":"Goteborg"},"Street":{"S":"Slanbarsvagen"},"Number":{"S":"34"},"ZipCode":{"S":"41787"}}	Express	2020-06-22T08:20:00	w#12376
	sh#98765	EntityType	Address	Type	Date	Warehouseld
		shipment	{"Country":{"S":"Sweden"},"County":{"S":"Vastra Gotaland"},"City":{"S":"Goteborg"},"Street":{"S":"Slanbarsvagen"},"Number":{"S":"34"},"ZipCode":{"S":"41787"}}	Express	2020-06-22T10:20:00	w#12345

Step 8: Address access pattern 9 (`getOrderByProductIdForDateRange`)

We created an *order* item collection in the previous step. This access pattern has new lookup dimensions (`ProductID` and `Date`) which requires you to scan the whole table and filter out relevant records to fetch targeted items. In order to address this access pattern, we'll need to create a global secondary index (GSI). Import AnOnlineShop_10.json to create a new item collection using the GSI that makes it possible to retrieve `orderItem` data from several *order* item collections. The data now has `GSI1-PK` and `GSI1-SK` which will be `GSI1`'s partition key and sort key, respectively.

DynamoDB automatically populates items which contain a GSI's key attributes from the table to the GSI. There is no need to manually do any additional inserts into the GSI.

To address access pattern 9, perform a query on `GSI1` with `GSI1-PK=productId` and `GSI1SK between (date1, date2)`.

Base table:

Primary key		Attributes				
Partition key: PK	**Sort key: SK**					
o#12345	p#12345	EntityType	GSI1-PK	GSI1-SK	Price	Quantity
		orderItem	p#12345	2020-06-21T19:18:00	100	2
	p#99887	EntityType	GSI1-PK	GSI1-SK	Price	Quantity
		orderItem	p#99887	2020-06-21T19:20:00	40	5

GSI1:

Primary key		Attributes				
Partition key: GSI1-PK	**Sort key: GSI1-SK**					
p#12345	2020-06-21T19:18:00	PK	SK	EntityType	Quantity	Price
		o#12345	p#12345	orderItem	2	100
p#99887	2020-06-21T19:20:00	PK	SK	EntityType	Quantity	Price
		o#12345	p#99887	orderItem	5	40

Step 9: Address access patterns 10 (`getInvoiceByInvoiceId`) and 11 (`getPaymentByInvoiceId`)

Import AnOnlineShop_11.json to address access patterns 10 (`getInvoiceByInvoiceId`) and 11 (`getPaymentByInvoiceId`), both of which are related to `invoice`. Even though these are two different access patterns, they are realized using the same key condition. `Payments` are defined as an attribute with the map data type on the `invoice` entity.

> **Note**
>
> `GSI1-PK` and `GSI1-SK` is overloaded to store information about different entities so that multiple access patterns can be served from the same GSI. For more information about GSI overloading, see Overloading Global Secondary Indexes in DynamoDB.

To address access pattern 10 and 11, query `GSI1` with `GSI1-PK=invoiceId` and `GSI1-SK=invoiceId`.

GSI1:

Primary key		Attributes							
Partition key: GSI1-PK	Sort key: GSI1-SK	PK	SK	EntityType	GSI2-PK	GSI2-SK	Detail	Amount	Date
i#55443	i#55443	o#12345	i#55443	invoice	c#12345	i#2020-06-21T19:18:00	{"Payments":{"L":[{"M":{"Type":{"S":"GiftCard"},"Amount":{"N":"100"},"Data":{"S":"GiftCard data here..."}}},{"M":{"Type":{"S":"MasterCard"},"Amount":{"N":"300"},"Data":{"S":"Payment data here..."}}}]}}	400	2020-06-21T19:18:00

Step 10: Address access patterns 12 (`getShipmentDetailsByShipmentId`) and 13 (`getShipmentByWarehouseId`)

Import AnOnlineShop_12.json to address access patterns 12 (`getShipmentDetailsByShipmentId`) and 13 (`getShipmentByWarehouseId`).

Notice that `shipmentItem` entities are added to the *order* item collection on the base table in order to be able to retrieve all details about an order in a single query operation.

Base table:

Primary key		Attributes							
Partition key: PK	Sort key: SK								
o#12345	sh#88899	EntityType	GSI1-PK	GSI1-SK	GSI2-PK	GSI2-SK	Address	Type	Date
		shipment	sh#88899	sh#88899	w#12376	sh#88899	{"Country": {"S":"Sweden"},"County": {"S":"Vastra Gotaland"},"City": {"S":"Goteborg"},"Street": {"S":"Slanbarsvagen"},"Number": {"S":"34"},"ZipCode": {"S":"41787"}}	Express	2020-06-22T08:20:00
	sh#98765	EntityType	GSI1-PK	GSI1-SK	GSI2-PK	GSI2-SK	Address	Type	Date
		shipment	sh#98765	sh#98765	w#12345	sh#98765	{"Country": {"S":"Sweden"},"County": {"S":"Vastra Gotaland"},"City": {"S":"Goteborg"},"Street": {"S":"Slanbarsvagen"},"Number": {"S":"34"},"ZipCode": {"S":"41787"}}	Express	2020-06-22T10:20:00
	shp#12345	EntityType	GSI1-PK	GSI1-SK	Quantity				
		shipmentItem	sh#98765	p#99887	3				
	shp#54321	EntityType	GSI1-PK	GSI1-SK	Quantity				
		shipmentItem	sh#88899	p#99887	2				
	shp#55555	EntityType	GSI1-PK	GSI1-SK	Quantity				
		shipmentItem	sh#98765	p#12345	2				

The `GSI1` partition and sort keys have already been used to model a one-to-many relationship between `shipment` and `shipmentItem`. To address access pattern 12 (`getShipmentDetailsByShipmentId`), query `GSI1` with `GSI1-PK=shipmentId` and `GSI1-SK=shipmentId`.

GSI1:

Primary key		Attributes							
Partition key: GSI1-PK	Sort key: GSI1-SK								
	p#99887	PK	SK	EntityType	Quantity				
		o#12345	shp#54321	shipmentItem	2				
sh#88899	sh#88899	PK	SK	EntityType	GSI2-PK	GSI2-SK	Address	Type	Date
		o#12345	sh#88899	shipment	w#12376	sh#88899	{"Country":{"S":"Sweden"},"County":{"S":"Vastra Gotaland"},"City":{"S":"Goteborg"},"Street":{"S":"Slanbarsvagen"},"Number":{"S":"34"},"ZipCode":{"S":"41787"}}	Express	2020-06-22T08:20:00
sh#98765	p#12345	PK	SK	EntityType	Quantity				
		o#12345	shp#55555	shipmentItem	2				
	p#99887	PK	SK	EntityType	Quantity				
		o#12345	shp#12345	shipmentItem	3				
	sh#98765	PK	SK	EntityType	GSI2-PK	GSI2-SK	Address	Type	Date
		o#12345	sh#98765	shipment	w#12345	sh#98765	{"Country":{"S":"Sweden"},"County":{"S":"Vastra Gotaland"},"City":{"S":"Goteborg"},"Street":{"S":"Slanbarsvagen"},"Number":{"S":"34"},"ZipCode":{"S":"41787"}}	Express	2020-06-22T10:20:00

We'll need to create another GSI (`GSI2`) to model the new one-to-many relationship between `warehouse` and `shipment` for access pattern 13 (`getShipmentByWarehouseId`). To address this access pattern, query `GSI2` with `GSI2-PK=warehouseId` and `GSI2-SK begins_with "sh#"`.

GSI2:

Primary key		Attributes							
Partition key: GSI2-PK	Sort key: GSI2-SK								
		PK	SK	EntityType	GSI1-PK	GSI1-SK	Address	Type	Date
w#12376	sh#88899	o#12345	sh#88899	shipment	sh#88899	sh#88899	{"Country": {"S":"Sweden"},"County": {"S":"Vastra Gotaland"},"City": {"S":"Goteborg"},"Street": {"S":"Slanbarsvagen"},"Number": {"S":"34"},"ZipCode": {"S":"41787"}}	Express	2020-06-22T08:20:00
		PK	SK	EntityType	GSI1-PK	GSI1-SK	Address	Type	Date
w#12345	sh#98765	o#12345	sh#98765	shipment	sh#98765	sh#98765	{"Country": {"S":"Sweden"},"County": {"S":"Vastra Gotaland"},"City": {"S":"Goteborg"},"Street": {"S":"Slanbarsvagen"},"Number": {"S":"34"},"ZipCode": {"S":"41787"}}	Express	2020-06-22T10:20:00

Step 11: Address access patterns 14 (`getProductInventoryByWarehouseId`) 15 (`getInvoiceByCustomerIdForDateRange`), and 16 (`getProductsByCustomerIdForDateRange`)

Import AnOnlineShop_13.json to add data related to the next set of access patterns. To address access pattern 14 (`getProductInventoryByWarehouseId`), query `GSI2` with `GSI2-PK=warehouseId` and `GSI2-SK begins_with “p#”`.

GSI2:

Primary key		Attributes							
Partition key: GSI2-PK	**Sort key: GSI2-SK**								
w#12345	p#12345	PK	SK	EntityType	Quantity				
		p#12345	w#12345	warehouseItem	50				
	p#99887	PK	SK	EntityType	Quantity				
		p#99887	w#12345	warehouseItem	4				
c#12345	i#2020-06-21T19:18:00	PK	SK	EntityType	GSI1-PK	GSI1-SK	Detail	Amount	Date
		o#12345	i#55443	invoice	i#55443	i#55443	{"Payments":{"L":[{"M":{"Type":{"S":"GiftCard"},"Amount":{"N":"100"},"Data":{"S":"GiftCard data here..."}}},{"M":{"Type":{"S":"MasterCard"},"Amount":{"N":"300"},"Data":{"S":"Payment data here..."}}}]}}	400	2020-06-21T19:18:00
	p#2020-06-21T19:18:00	PK	SK	EntityType	GSI1-PK	GSI1-SK	Price	Quantity	
		o#12345	p#12345	orderItem	p#12345	2020-06-21T19:18:00	100	2	
	p#2020-06-21T19:20:00	PK	SK	EntityType	GSI1-PK	GSI1-SK	Price	Quantity	
		o#12345	p#99887	orderItem	p#99887	2020-06-21T19:20:00	40	5	

To address access pattern 15 (`getInvoiceByCustomerIdForDateRange`), query `GSI2` with `GSI2-PK=customerId` and `GSI2-SK between (i#date1, i#date2)`.

GSI2:

Primary key		Attributes							
Partition key: GSI2-PK	Sort key: GSI2-SK								
w#12345	p#12345	PK	SK	EntityType	Quantity				
		p#12345	w#12345	warehouseItem	50				
	p#99887	PK	SK	EntityType	Quantity				
		p#99887	w#12345	warehouseItem	4				
c#12345	i#2020-06-21T19:18:00	PK	SK	EntityType	GSI1-PK	GSI1-SK	Detail	Amount	Date
		o#12345	i#55443	invoice	i#55443	i#55443	{"Payments":{"L":[{"M":{"Type":{"S":"GiftCard"},"Amount":{"N":"100"},"Data":{"S":"GiftCard data here..."}}},{"M":{"Type":{"S":"MasterCard"},"Amount":{"N":"300"},"Data":{"S":"Payment data here..."}}}]}}	400	2020-06-21T19:18:00
	p#2020-06-21T19:18:00	PK	SK	EntityType	GSI1-PK	GSI1-SK	Price	Quantity	
		o#12345	p#12345	orderItem	p#12345	2020-06-21T19:18:00	100	2	
	p#2020-06-21T19:20:00	PK	SK	EntityType	GSI1-PK	GSI1-SK	Price	Quantity	
		o#12345	p#99887	orderItem	p#99887	2020-06-21T19:20:00	40	5	

To address access pattern 16 (`getProductsByCustomerIdForDateRange`), query `GSI2` with `GSI2-PK=customerId` and `GSI2-SK between (p#date1, p#date2)`.

GSI2:

Primary key		Attributes							
Partition key: GSI2-PK	Sort key: GSI2-SK								
w#12345	p#12345	PK	SK	EntityType	Quantity				
		p#12345	w#12345	warehouseItem	50				
	p#99887	PK	SK	EntityType	Quantity				
		p#99887	w#12345	warehouseItem	4				
c#12345	i#2020-06-21T19:18:00	PK	SK	EntityType	GSI1-PK	GSI1-SK	Detail	Amount	Date
		o#12345	i#55443	invoice	i#55443	i#55443	{"Payments":{"L":[{"M":{"Type":{"S":"GiftCard"},"Amount":{"N":"100"},"Data":{"S":"GiftCard data here..."}}},{"M":{"Type":{"S":"MasterCard"},"Amount":{"N":"300"},"Data":{"S":"Payment data here..."}}}]}}	400	2020-06-21T19:18:00
	p#2020-06-21T19:18:00	PK	SK	EntityType	GSI1-PK	GSI1-SK	Price	Quantity	
		o#12345	p#12345	orderItem	p#12345	2020-06-21T19:18:00	100	2	
	p#2020-06-21T19:20:00	PK	SK	EntityType	GSI1-PK	GSI1-SK	Price	Quantity	
		o#12345	p#99887	orderItem	p#99887	2020-06-21T19:20:00	40	5	

Note

In NoSQL Workbench, *facets* represent an application's different data access patterns for DynamoDB. Facets give you a way to view a subset of the data in a table, without having to see records that don't meet the constraints of the facet. Facets are considered a visual data modeling tool, and don't exist as a usable construct in DynamoDB as they are purely an aid for modeling access patterns.

Import AnOnlineShop_facets.json to see the facets for this use case.

All access patterns and how the schema design addresses them are summarized in the table below:

Access pattern	Base table/GSI/LSI	Operation	Partition key value	Sort key value
getCustomerByCustomerId	Base table	GetItem	PK=customerId	SK=customerId
getProductByProductId	Base table	GetItem	PK=productId	SK=productId
getWarehouseByWarehouseId	Base table	GetItem	PK=warehouseId	SK=warehouseId
getProductInventoryByProductId	Base table	Query	PK=productId	SK begins_with "w#"
getOrderDetailsByOrderId	Base table	Query	PK=orderId	
getProductByOrderId	Base table	Query	PK=orderId	SK begins_with "p#"
getInvoiceByOrderId	Base table	Query	PK=orderId	SK begins_with "i#"
getShipmentByOrderId	Base table	Query	PK=orderId	SK begins_with "sh#"
getOrderByProductIdForDateRange	GSI1	Query	PK=productId	SK between date1 and date2
getInvoiceByInvoiceId	GSI1	Query	PK=invoiceId	SK=invoiceId

Access pattern	Base table/GSI/LSI	Operation	Partition key value	Sort key value
getPaymentByInvoiceId	GSI1	Query	PK=invoiceId	SK=invoiceId
getShipmentDetailsByShipmentId	GSI1	Query	PK=shipmentId	SK=shipmentId
getShipmentByWarehouseId	GSI2	Query	PK=warehouseId	SK begins_with "sh#"
getProductInventoryByWarehouseId	GSI2	Query	PK=warehouseId	SK begins_with "p#"
getInvoiceByCustomerIdForDateRange	GSI2	Query	PK=customerId	SK between i#date1 and i#date2
getProductsByCustomerIdForDateRange	GSI2	Query	PK=customerId	SK between p#date1 and p#date2

Online shop final schema

Here are the final schema designs. To download this schema design as a JSON file, see DynamoDB Design Patterns on GitHub.

Base table

Primary key		Attributes			
Partition key: PK	**Sort key: SK**				
c#12345	c#12345	EntityType	Email	Name	
		customer	samaneh@example.com	Samaneh	
c#23456	c#23456	EntityType	Email	Name	
		customer	kathleen@example.com	Kathleen	
c#54321	c#54321	EntityType	Email	Name	
		customer	henrik@example.com	Henrik	
p#12345	p#12345	EntityType	Detail	Price	
		product	{"Name":{"S":"Options Open"},"Description":{"S":"The latest album"}}	100	
	w#12345	EntityType	GSI2-PK	GSI2-SK	Quantity
		warehouseItem	w#12345	p#12345	50
p#99887	p#99887	EntityType	Detail	Price	
		product	{"Name":{"S":"The Book"},"Description":{"S":"The best book ever"}}	40	
	w#12345	EntityType	GSI2-PK	GSI2-SK	Quantity
		warehouseItem	w#12345	p#99887	4
	w#12376	EntityType	Quantity		
		warehouseItem	4		
w#12345	w#12345	EntityType	Address		
		warehouse	{"Country":{"S":"Sweden"},"County":{"S":"Vastra Gotaland"},"City":{"S":"Goteborg"},"Street":{"S":"MainStreet"},"Number":{"S":"20"},"ZipCode":{"S":"41111"}}		
w#12376	w#12376	EntityType	Address		
		warehouse	{"Country":{"S":"Sweden"},"County":		

GSI1

Primary key		Attributes							
Partition key: GSI1-PK	**Sort key: GSI1-SK**								
p#12345	2020-06-21T19:18:00	PK	SK	EntityType	GSI2-PK	GSI2-SK	Price	Quantity	
		o#12345	p#12345	orderItem	c#12345	2020-06-21T19:18:00	100	2	
p#99887	2020-06-21T19:20:00	PK	SK	EntityType	GSI2-PK	GSI2-SK	Price	Quantity	
		o#12345	p#99887	orderItem	c#12345	2020-06-21T19:20:00	40	5	
i#55443	i#55443	PK	SK	EntityType	GSI2-PK	GSI2-SK	Detail	Amount	Date
		o#12345	i#55443	invoice	c#12345	2020-06-21T19:18:00	{"Payments":{"L":[{"M":{"Type":{"S":"GiftCard"},"Amount":{"N":"100"},"Data":{"S":"GiftCard data here..."}}},{"M":{"Type":{"S":"MasterCard"},"Amount":{"N":"300"},"Data":{"S":"Payment data here..."}}}]}}	400	2020-06-21T19:18:00
sh#88899	p#99887	PK	SK	EntityType	Quantity				
		o#12345	shp#54321	shipmentItem	2				
	sh#88899	PK	SK	EntityType	GSI2-PK	GSI2-SK	Address	Type	Date
		o#12345	sh#88899	shipment	w#12376	sh#88899	{"Country":{"S":"Sweden"},"County":{"S":"Vastra Gotaland"},"City":{"S":"Goteborg"},"Street":{"S":"Slanbarsvagen"},"Number":{"S":"34"},"ZipCode":{"S":"41787"}}	Express	2020-06-22T08:20:00
sh#98765	p#12345	PK	SK	EntityType	Quantity				
		o#12345	shp#55555	shipmentItem	2				
	p#99887	PK	SK	EntityType	Quantity				
		o#12345	shp#12345	shipmentItem	3				
	sh#98765	PK	SK	EntityType	GSI2-PK	GSI2-SK	Address	Type	Date
		o#12345	sh#98765	shipment	w#12345	sh#98765	{"Country":{"S":"Sweden"},"County":{"S":"Vastra Gotaland"},"City":{"S":"Goteborg"},"Street":{"S":"Slanbarsvagen"},"Number":{"S":"34"},"ZipCode":{"S":"41787"}}	Express	2020-06-22T10:20:00

GSI2

Primary key		Attributes							
Partition key: GSI2-PK	**Sort key: GSI2-SK**								
w#12345	p#12345	PK	SK	EntityType	Quantity				
		p#12345	w#12345	warehouseItem	50				
	p#99887	PK	SK	EntityType	Quantity				
		p#99887	w#12345	warehouseItem	4				
	sh#98765	PK	SK	EntityType	GSI1-PK	GSI1-SK	Address	Type	Date
		o#12345	sh#98765	shipment	sh#98765	sh#98765	{"Country":{"S":"Sweden"},"County":{"S":"Vastra Gotaland"},"City":{"S":"Goteborg"},"Street":{"S":"Slanbarsvagen"},"Number":{"S":"34"},"ZipCode":{"S":"41787"}}	Express	2020-06-22T10:20:00
c#12345	2020-06-21T19:18:00	PK	SK	EntityType	GSI1-PK	GSI1-SK	Price	Quantity	
		o#12345	p#12345	orderItem	p#12345	2020-06-21T19:18:00	100	2	
	2020-06-21T19:18:00	PK	SK	EntityType	GSI1-PK	GSI1-SK	Detail	Amount	Date
		o#12345	i#55443	invoice	i#55443	i#55443	{"Payments":{"L":[{"M":{"Type":{"S":"GiftCard"},"Amount":{"N":"100"},"Data":{"S":"GiftCard data here..."}}},{"M":{"Type":{"S":"MasterCard"},"Amount":{"N":"300"},"Data":{"S":"Payment data here..."}}}]}}	400	2020-06-21T19:18:00
	2020-06-21T19:20:00	PK	SK	EntityType	GSI1-PK	GSI1-SK	Price	Quantity	
		o#12345	p#99887	orderItem	p#99887	2020-06-21T19:20:00	40	5	
w#12376	sh#88899	PK	SK	EntityType	GSI1-PK	GSI1-SK	Address	Type	Date
		o#12345	sh#88899	shipment	sh#88899	sh#88899	{"Country":{"S":"Sweden"},"County":{"S":"Vastra Gotaland"},"City":{"S":"Goteborg"},"Street":{"S":"Slanbarsvagen"},"Number":{"S":"34"},"ZipCode":{"S":"41787"}}	Express	2020-06-22T08:20:00

Using NoSQL Workbench with this schema design

You can import this final schema into NoSQL Workbench, a visual tool that provides data modeling, data visualization, and query development features for DynamoDB, to further explore and edit your new project. Follow these steps to get started:

1. Download NoSQL Workbench. For more information, see the section called “Download”.
2. Download the JSON schema file listed above, which is already in the NoSQL Workbench model format.
3. Import the JSON schema file into NoSQL Workbench. For more information, see the section called “Importing an existing model”.
4. Once you've imported into NOSQL Workbench, you can edit the data model. For more information, see the section called “Editing an existing model”.

Best practices for modeling relational data in DynamoDB

This section provides best practices for modeling relational data in Amazon DynamoDB. First, we introduce traditional data modeling concepts. Then, we describe the advantages of using DynamoDB over traditional relational database management systems—how it eliminates the need for JOIN operations and reduces overhead.

We then explain how to design a DynamoDB table that scales efficiently. Finally, we provide an example of how to model relational data in DynamoDB.

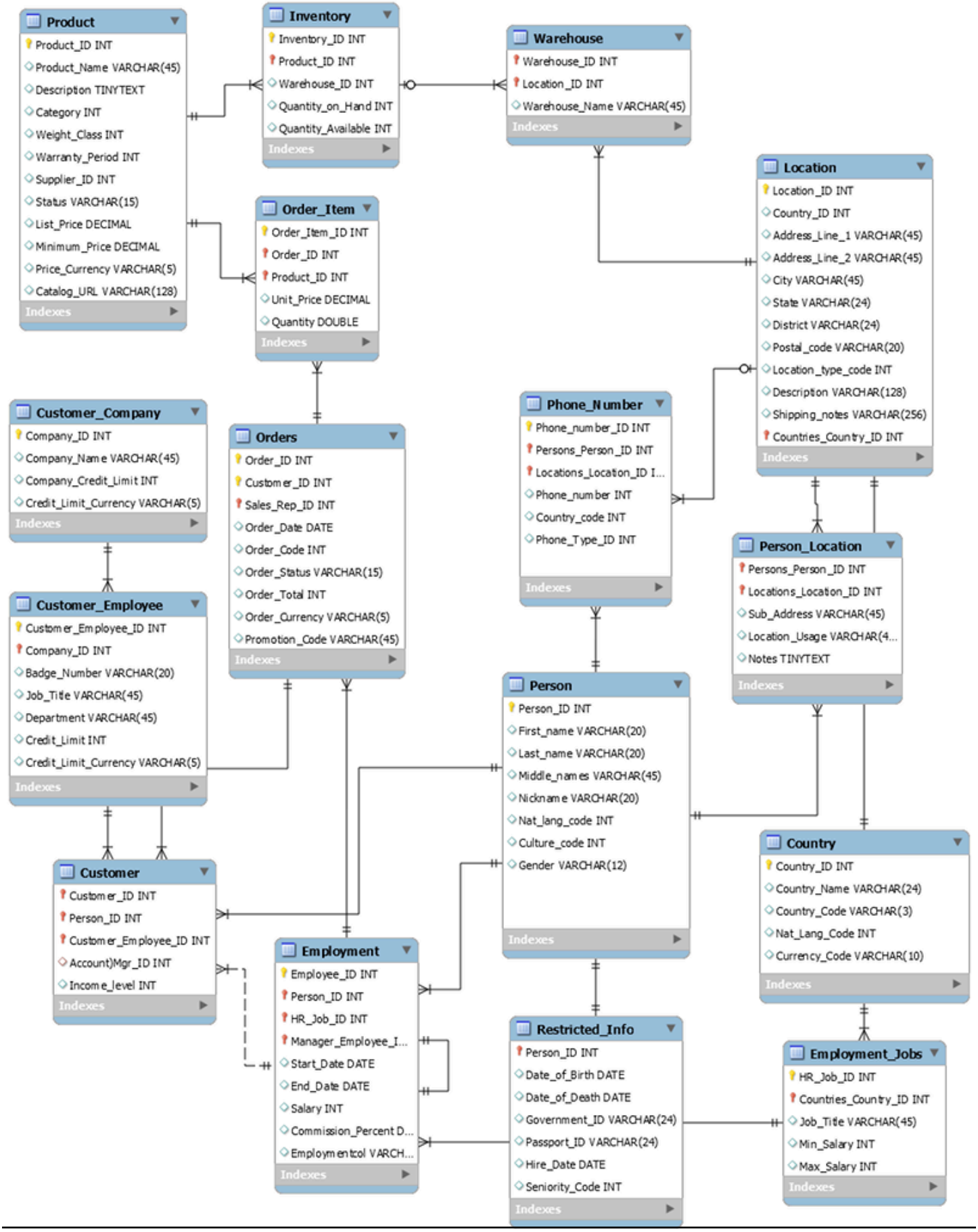
Topics

- Traditional relational database models
- How DynamoDB eliminates the need for JOIN operations
- How DynamoDB transactions eliminate overhead to the write process
- First steps for modeling relational data in DynamoDB
- Example of modeling relational data in DynamoDB

Traditional relational database models

A traditional relational database management system (RDBMS) stores data in a normalized relational structure. The objective of the relational data model is to reduce the duplication of data (through normalization) to support referential integrity and reduce data anomalies.

The following schema is an example of a relational data model for a generic order-entry application. The application supports a human resources schema that backs the operational and business support systems of a theoretical manufacturer.


Product
Product_ID INT
Product_Name VARCHAR(45)
Description TINYTEXT
Category INT
Weight_Class INT
Warranty_Period INT
Supplier_ID INT
Status VARCHAR(15)
List_Price DECIMAL
Minimum_Price DECIMAL
Price_Currency VARCHAR(5)
Catalog_URL VARCHAR(128)
Indexes
Inventory
Inventory_ID INT
Product_ID INT
Warehouse_ID INT
Quantity_on_Hand INT
Quantity_Available INT
Indexes
Warehouse
Warehouse_ID INT
Location_ID INT
Warehouse_Name VARCHAR(45)
Indexes
Location
Location_ID INT
Country_ID INT
Address_Line_1 VARCHAR(45)
Address_Line_2 VARCHAR(45)
City VARCHAR(45)
State VARCHAR(24)
District VARCHAR(24)
Postal_code VARCHAR(20)
Location_type_code INT
Description VARCHAR(128)
Shipping_notes VARCHAR(256)
Countries_Country_ID INT
Indexes
Order_Item
Order_Item_ID INT
Order_ID INT
Product_ID INT
Unit_Price DECIMAL
Quantity DOUBLE
Indexes
Customer_Company
Company_ID INT
Company_Name VARCHAR(45)
Company_Credit_Limit INT
Credit_Limit_Currency VARCHAR(5)
Indexes
Orders
Order_ID INT
Customer_ID INT
Sales_Rep_ID INT
Order_Date DATE
Order_Code INT
Order_Status VARCHAR(15)
Order_Total INT
Order_Currency VARCHAR(5)
Promotion_Code VARCHAR(45)
Indexes
Phone_Number
Phone_number_ID INT
Persons_Person_ID INT
Locations_Location_ID I...
Phone_number INT
Country_code INT
Phone_Type_ID INT
Indexes
Person_Location
Persons_Person_ID INT
Locations_Location_ID INT
Sub_Address VARCHAR(45)
Location_Usage VARCHAR(4...
Notes TINYTEXT
Indexes
Customer_Employee
Customer_Employee_ID INT
Company_ID INT
Badge_Number VARCHAR(20)
Job_Title VARCHAR(45)
Department VARCHAR(45)
Credit_Limit INT
Credit_Limit_Currency VARCHAR(5)
Indexes
Person
Person_ID INT
First_name VARCHAR(20)
Last_name VARCHAR(20)
Middle_names VARCHAR(45)
Nickname VARCHAR(20)
Nat_lang_code INT
Culture_code INT
Gender VARCHAR(12)
Indexes
Country
Country_ID INT
Country_Name VARCHAR(24)
Country_Code VARCHAR(3)
Nat_Lang_Code INT
Currency_Code VARCHAR(10)
Indexes
Customer
Customer_ID INT
Person_ID INT
Customer_Employee_ID INT
Account)Mgr_ID INT
Income_level INT
Indexes
Employment
Employee_ID INT
Person_ID INT
HR_Job_ID INT
Manager_Employee_I...
Start_Date DATE
End_Date DATE
Salary INT
Commission_Percent D...
Employmentcol VARCH...
Indexes
Restricted_Info
Person_ID INT
Date_of_Birth DATE
Date_of_Death DATE
Government_ID VARCHAR(24)
Passport_ID VARCHAR(24)
Hire_Date DATE
Seniority_Code INT
Indexes
Employment_Jobs
HR_Job_ID INT
Countries_Country_ID INT
Job_Title VARCHAR(45)
Min_Salary INT
Max_Salary INT
Indexes

As a non-relational database service, DynamoDB offers many advantages over traditional relational database management systems.

How DynamoDB eliminates the need for JOIN operations

An RDBMS uses a structure query language (SQL) to return data to the application. Because of the normalization of the data model, such queries typically require the use of the `JOIN` operator to combine data from one or more tables.

For example, to generate a list of purchase order items sorted by the quantity in stock at all warehouses that can ship each item, you could issue the following SQL query against the preceding schema.

```
SELECT * FROM Orders
  INNER JOIN Order_Items ON Orders.Order_ID = Order_Items.Order_ID
  INNER JOIN Products ON Products.Product_ID = Order_Items.Product_ID
  INNER JOIN Inventories ON Products.Product_ID = Inventories.Product_ID
  ORDER BY Quantity_on_Hand DESC
```

SQL queries of this kind can provide a flexible API for accessing data, but they require a significant amount of processing. Each join in the query increases the runtime complexity of the query because the data for each table must stage and then be assembled to return the result set.

Additional factors that can impact how long it takes the queries to run are the size of the tables and whether the columns being joined have indexes. The preceding query initiates complex queries across several tables and then sorts the result set.

Eliminating the need for `JOIN`s is at the heart of NoSQL data modeling. This is why we built DynamoDB to support Amazon.com, and why DynamoDB can deliver consistent performance at any scale. Given the runtime complexity of SQL queries and `JOIN`s, RDBMS performance is not constant at scale. This causes performance issues as customer applications grow.

While normalizing data does reduce the amount of data stored to disk, often the most constrained resources that impact performance are CPU time and network latency.

DynamoDB is built to minimize both constraints by eliminating `JOIN`s (and encouraging denormalization of data) and optimizing the database architecture to fully answer an application query with a single request to an item. These qualities enable DynamoDB to provide single-digit, millisecond performance at any scale. This is because the runtime complexity for DynamoDB operations is constant, regardless of data size, for common access patterns.

How DynamoDB transactions eliminate overhead to the write process

Another factor that can slow down an RDBMS is the use of transactions to write to a normalized schema. As shown in the example, relational data structures used by most online transaction processing (OLTP) applications must be broken down and distributed across multiple logical tables when they are stored in an RDBMS.

Therefore, an ACID-compliant transaction framework is necessary to avoid race conditions and data integrity issues that could occur if an application tries to read an object that is in the process of being written. Such a transaction framework, when coupled with a relational schema, can add significant overhead to the write process.

The implementation of transactions in DynamoDB prohibits common scaling issues that are found with an RDBMS. DynamoDB does this by issuing a transaction as a single API call and bounding the number of items that can be accessed in that single transaction. Long-running transactions can cause operational issues by holding locks on the data either for a long time, or perpetually, because the transaction is never closed.

To prevent such issues in DynamoDB, transactions were implemented with two distinct API operations: `TransactWriteItems` and `TransactGetItems`. These API operations do not have begin and end semantics that are common in an RDBMS. Further, DynamoDB has a 100-item access limit within a transaction to similarly prevent long-running transactions. To learn more about DynamoDB transactions, see Working with transactions.

For these reasons, when your business requires a low-latency response to high-traffic queries, taking advantage of a NoSQL system generally makes technical and economic sense. Amazon DynamoDB helps solve the problems that limit relational system scalability by avoiding them.

The performance of an RDBMS does not typically scale well for the following reasons:

- It uses expensive joins to reassemble required views of query results.
- It normalizes data and stores it on multiple tables that require multiple queries to write to disk.
- It generally incurs the performance costs of an ACID-compliant transaction system.

DynamoDB scales well for these reasons:

- Schema flexibility lets DynamoDB store complex hierarchical data within a single item.
- Composite key design lets it store related items close together on the same table.

- Transactions are performed in a single operation. The limit for the number of items that can be accessed is 100, to avoid long-running operations.

Queries against the data store become much simpler, often in the following form:

```
SELECT * FROM Table_X WHERE Attribute_Y = "somevalue"
```

DynamoDB does far less work to return the requested data compared to the RDBMS in the earlier example.

First steps for modeling relational data in DynamoDB

> **Note**
>
> NoSQL design requires a different mindset than RDBMS design. For an RDBMS, you can create a normalized data model without thinking about access patterns. You can then extend it later when new questions and query requirements arise. By contrast, in Amazon DynamoDB, you shouldn't start designing your schema until you know the questions that it needs to answer. Understanding the business problems and the application use cases up front is absolutely essential.

To start designing a DynamoDB table that will scale efficiently, you must take several steps first to identify the access patterns that are required by the operations and business support systems (OSS/BSS) that it needs to support:

- For new applications, review user stories about activities and objectives. Document the various use cases you identify, and analyze the access patterns that they require.
- For existing applications, analyze query logs to find out how people are currently using the system and what the key access patterns are.

After completing this process, you should end up with a list that might look something like the following.

Access Patterns for Order Entry Application

Pattern #	Access Pattern Description
1	Look up Employee Details by Employee ID
2	Query Employee Details by Employee Name
3	Find an Employee's Phone Number(s)
4	Find a Customer's Phone Number(s)
5	Get Orders for Customer within Date Range
6	Show all Open Orders within Date Range
7	See all Employees hired recently
8	Find all Employees in Warehouse
9	Get all Items on Order for Product
10	Get Inventories for Product at all Warehouses
11	Get Customers by Account Rep
12	Get Orders by Account Rep
13	Get Employees with Job Title
14	Get Inventory by Product and Warehouse
15	Get Total Product Inventory

In a real application, your list might be much longer. But this collection represents the range of query pattern complexity that you might find in a production environment.

A modern approach to DynamoDB schema design uses aggregate-oriented principles, grouping data based on access patterns rather than rigid entity boundaries. This approach considers multiple design patterns:

- *Single Table Design* - Using composite sort keys, overloaded global secondary indexes, and adjacency list patterns to store multiple entity types in one table
- *Multi-Table Design* - Using separate tables for entities with independent operational characteristics and low access correlation, with strategic GSIs for cross-entity queries
- *Aggregate Design* - Embedding related data when always accessed together (Order + OrderItems) or using item collections for identifying relationships (Product + Inventory)

The choice between these approaches depends on your specific access patterns, data characteristics, and operational requirements. You can use these elements to structure the data so that an application can retrieve whatever it needs for a given access pattern using a single query on a table or index.

> **Note**
>
> The choice between single-table and multi-table design depends on your specific requirements. Single-table design works well when entities have high access correlation and similar operational characteristics. Multi-table design is preferred when entities have independent operational requirements, different access patterns, or when you need clear operational boundaries. The example in this guide demonstrates a multi-table approach with strategic aggregation and denormalization.

To use NoSQL Workbench for DynamoDB to help visualize your partition key design, see Building data models with NoSQL Workbench.

Example of modeling relational data in DynamoDB

This example describes how to model relational data in Amazon DynamoDB. The DynamoDB table design corresponds to the relational order entry schema that is shown in Relational modeling. This design uses multiple specialized tables rather than a single adjacency list, providing clear operational boundaries while leveraging strategic GSIs to serve all access patterns efficiently.

The design approach uses aggregate-oriented principles, grouping data based on access patterns rather than rigid entity boundaries. Key design decisions include using separate tables for entities with low access correlation, embedding related data when always accessed together, and using item collections for identifying relationships.

The following tables and their accompanying indexes support the relational order entry schema:

Employee Table Design

The Employee table stores employee information as a single entity per item, optimized for direct employee lookups and supporting multiple query patterns through strategic GSIs. This table demonstrates the principle of designing separate tables for entities with independent operational characteristics and low cross-entity access correlation.

The table uses a simple partition key (employee_id) without a sort key, as each employee is a distinct entity. Four GSIs enable efficient querying by different attributes:

- *EmployeeByName GSI* - Uses INCLUDE projection with all employee attributes to support complete employee detail retrieval by name, handling potential duplicate names with employee_id as sort key
- *EmployeeByWarehouse GSI* - Uses INCLUDE projection with only essential attributes (name, job_title, hire_date) to minimize storage costs while supporting warehouse-based queries
- *EmployeeByJobTitle GSI* - Enables role-based queries with INCLUDE projection for reporting and organizational analysis
- *EmployeeByHireDate GSI* - Uses a static partition key value "EMPLOYEE" with hire_date as sort key to enable efficient date range queries for recent hires. Since employee additions/updates are typically under 1,000 WCU, a single partition can handle the write load without hot partition issues

Employee Table - Base Table Structure

employee_id (PK)	name	phone_numbers	warehouse_id	job_title	hire_date	entity_type
emp_001	John Smith	["+1-555-0101"]	wh_sea	Manager	2024-03-15	EMPLOYEE
emp_002	Jane Doe	["+1-555-0102", "+1-555-0103"]	wh_sea	Associate	2025-01-10	EMPLOYEE
emp_003	Bob Wilson	["+1-555-0104"]	wh_pdx	Associate	2025-06-20	EMPLOYEE

employee_id (PK)	name	phone_numbers	warehouse_id	job_title	hire_date	entity_type
emp_004	Alice Brown	["+1-555-0105"]	wh_pdx	Supervisor	2023-11-05	EMPLOYEE
emp_005	Charlie Davis	["+1-555-0106"]	wh_sea	Associate	2025-12-01	EMPLOYEE

EmployeeByName GSI - Supporting Employee Name Queries

name (GSI-PK)	employee_id (GSI-SK)	phone_numbers	warehouse_id	job_title	hire_date
Alice Brown	emp_004	["+1-555-0105"]	wh_pdx	Supervisor	2023-11-05
Bob Wilson	emp_003	["+1-555-0104"]	wh_pdx	Associate	2025-06-20
Charlie Davis	emp_005	["+1-555-0106"]	wh_sea	Associate	2025-12-01
Jane Doe	emp_002	["+1-555-0102", "+1-555-0103"]	wh_sea	Associate	2025-01-10
John Smith	emp_001	["+1-555-0101"]	wh_sea	Manager	2024-03-15

EmployeeByWarehouse GSI - Supporting Warehouse Queries

warehouse_id (GSI-PK)	employee_id (GSI-SK)	name	job_title	hire_date
wh_pdx	emp_003	Bob Wilson	Associate	2025-06-20

warehouse_id (GSI-PK)	employee_id (GSI-SK)	name	job_title	hire_date
wh_pdx	emp_004	Alice Brown	Supervisor	2023-11-05
wh_sea	emp_001	John Smith	Manager	2024-03-15
wh_sea	emp_002	Jane Doe	Associate	2025-01-10
wh_sea	emp_005	Charlie Davis	Associate	2025-12-01

EmployeeByJobTitle GSI - Supporting Job Title Queries

job_title (GSI-PK)	employee_id (GSI-SK)	name	warehouse_id	hire_date
Associate	emp_002	Jane Doe	wh_sea	2025-01-10
Associate	emp_003	Bob Wilson	wh_pdx	2025-06-20
Associate	emp_005	Charlie Davis	wh_sea	2025-12-01
Manager	emp_001	John Smith	wh_sea	2024-03-15
Supervisor	emp_004	Alice Brown	wh_pdx	2023-11-05

EmployeeByHireDate GSI - Supporting Recent Hire Queries

entity_type (GSI-PK)	hire_date (GSI-SK)	employee_id	name	warehouse_id
EMPLOYEE	2023-11-05	emp_004	Alice Brown	wh_pdx
EMPLOYEE	2024-03-15	emp_001	John Smith	wh_sea
EMPLOYEE	2025-01-10	emp_002	Jane Doe	wh_sea
EMPLOYEE	2025-06-20	emp_003	Bob Wilson	wh_pdx

entity_type (GSI-PK)	hire_date (GSI-SK)	employee_id	name	warehouse_id
EMPLOYEE	2025-12-01	emp_005	Charlie Davis	wh_sea

Customer Table Design

The Customer table maintains customer information with strategic denormalization of account_rep_id to enable efficient account representative queries. This design choice trades slight storage overhead for query performance, eliminating the need for joins between customer and account representative data.

The table supports multiple phone numbers per customer using a list attribute, demonstrating DynamoDB's schema flexibility. The single GSI enables account representative workflows:

- *CustomerByAccountRep GSI* - Uses INCLUDE projection with name and email attributes to support account rep customer management without requiring full customer record retrieval

Customer Table - Base Table Structure

customer_id (PK)	name	phone_numbers	email	account_rep_id
cust_001	Acme Corp	["+1-555-1001"]	contact@acme.com	rep_001
cust_002	TechStart Inc	["+1-555-1002", "+1-555-1003"]	info@techstart.com	rep_001
cust_003	Global Traders	["+1-555-1004"]	sales@globaltraders.com	rep_002
cust_004	BuildRight LLC	["+1-555-1005"]	orders@buildright.com	rep_002
cust_005	FastShip Co	["+1-555-1006"]	support@fastship.com	rep_003

CustomerByAccountRep GSI - Supporting Account Rep Queries

account_rep_id (GSI-PK)	customer_id (GSI-SK)	name	email
rep_001	cust_001	Acme Corp	contact@acme.com
rep_001	cust_002	TechStart Inc	info@techstart.com
rep_002	cust_003	Global Traders	sales@globaltraders.com
rep_002	cust_004	BuildRight LLC	orders@buildright.com
rep_003	cust_005	FastShip Co	support@fastship.com

Order Table Design

The Order table uses vertical partitioning with separate items for order headers and order items. This design enables efficient product-based queries while maintaining all order components within the same partition for efficient access. Each order consists of multiple items:

- *Order Header* - Contains order metadata with PK=order_id, SK=order_id
- *Order Items* - Individual line items with PK=order_id, SK=product_id, enabling direct product queries

> **Note**
>
> This vertical partitioning approach trades the simplicity of embedded order items for enhanced query flexibility. Each order item becomes a separate DynamoDB item, enabling efficient product-based queries while maintaining all order data within the same partition for efficient retrieval in a single request.

The table includes strategic denormalization of account_rep_id (duplicated from Customer table) to enable direct account representative queries without requiring customer lookups. For high-

throughput write scenarios, OPEN orders include status and shard attributes to enable write sharding across multiple partitions.

Four GSIs support different query patterns with optimized projections:

- *OrderByCustomerDate GSI* - Uses INCLUDE projection with order summary and item details to support customer order history with date range filtering
- *OpenOrdersByDate GSI (Sparse, Sharded)* - Uses multi-attribute partition key (status + shard) with 5 shards to distribute 5,000 WPS (writes per second) across partitions (1,000 WPS each, matching DynamoDB's 1,000 WCU per partition limit). Only indexes OPEN orders (20% of total), which can help reduce GSI storage costs. Requires parallel queries across all 5 shards with client-side result merging
- *OrderByAccountRep GSI* - Uses INCLUDE projection with order summary attributes to support account representative workflows without full order details
- *ProductInOrders GSI* - Created from OrderItem records (PK=order_id, SK=product_id), this GSI enables queries to find all orders containing a specific product. Uses INCLUDE projection with order context (customer_id, order_date, quantity) for product demand analysis

Order Table - Base Table Structure (Vertical Partitioning)

PK	SK	customer_id	order_date	status	account_rep_id	quantity	price	shard
ord_001	ord_001	cust_001	2025-11-5	CLOSED	rep_001			
ord_001	prod_100					5	25.00	
ord_002	ord_002	cust_001	2025-12-0	OPEN	rep_001			0
ord_002	prod_101					10	15.00	
ord_003	ord_003	cust_002	2026-01-5	OPEN	rep_001			2
ord_003	prod_100					3	25.00	

OrderByCustomerDate GSI - Supporting Customer Order Queries

customer_id (GSI-PK)	order_date (GSI-SK)	order_id	status	total_amount	order_items	shard
cust_001	2025-11-15	ord_001	CLOSED	225.00	[{product_id: "prod_100", qty: 5}]	
cust_001	2025-12-20	ord_002	OPEN	150.00	[{product_id: "prod_101", qty: 10}]	0
cust_002	2026-01-05	ord_003	OPEN	175.00	[{product_id: "prod_100", qty: 3}]	2
cust_003	2025-10-10	ord_004	CLOSED	250.00	[{product_id: "prod_101", qty: 5}]	
cust_004	2026-01-03	ord_005	OPEN	200.00	[{product_id: "prod_100", qty: 20}]	1

OpenOrdersByDate GSI (Sparse, Sharded) - Supporting High-Throughput Open Order Queries

status (GSI-PK-1)	shard (GSI-PK-2)	order_date (SK)	order_id	customer_id	account_rep_id	order_items	total_amount
OPEN	0	2025-12-20	ord_002	cust_001	rep_001	[{product_id: "prod_101", qty: 10}]	150.00
OPEN	1	2026-01-03	ord_005	cust_004	rep_002	[{product_id: "prod_100", qty: 20}]	200.00
OPEN	2	2026-01-05	ord_003	cust_002	rep_001	[{product_id: "prod_100", qty: 3}]	175.00

OrderByAccountRep GSI - Supporting Account Rep Order Queries

account_rep_id (GSI-PK)	order_date (GSI-SK)	order_id	customer_id	status	total_amount
rep_001	2025-11-15	ord_001	cust_001	CLOSED	225.00
rep_001	2025-12-20	ord_002	cust_001	OPEN	150.00
rep_001	2026-01-05	ord_003	cust_002	OPEN	175.00
rep_002	2025-10-10	ord_004	cust_003	CLOSED	250.00
rep_002	2026-01-03	ord_005	cust_004	OPEN	200.00

ProductInOrders GSI - Supporting Product Order Queries

product_id (GSI-PK)	order_id (GSI-SK)	customer_id	order_date	quantity
prod_100	ord_001	cust_001	2025-11-15	5
prod_100	ord_003	cust_002	2026-01-05	3
prod_101	ord_002	cust_001	2025-12-20	10

Product Table Design

The Product table uses the item collection pattern to store both product metadata and inventory data within the same partition. This design leverages the identifying relationship between products and inventory - inventory cannot exist without a parent product. Using PK=product_id with SK=product_id for product metadata and SK=warehouse_id for inventory items eliminates the need for a separate Inventory table and GSI, reducing costs by approximately 50%.

This pattern enables efficient queries for both individual warehouse inventory (GetItem with composite key) and all warehouse inventory for a product (Query on partition key). The total_inventory attribute in the product metadata item provides denormalized aggregation for quick total inventory lookups.

Product Table - Base Table Structure (Item Collection Pattern)

product_id (PK)	warehouse_id (SK)	product_name	category	unit_price	inventory_quantity	total_inventory
prod_100	prod_100	Widget A	Hardware	25.00		500
prod_100	wh_sea				200	
prod_100	wh_pdx				150	
prod_100	wh_atl				150	
prod_101	prod_101	Gadget B	Electronics	50.00		300
prod_101	wh_sea				100	
prod_101	wh_pdx				200	

Each table is designed with specific Global Secondary Indexes (GSIs) to support the required access patterns efficiently. The design uses aggregate-oriented principles with strategic denormalization and sparse indexing to optimize both performance and cost.

Key design optimizations include:

- *Sparse GSI* - OpenOrdersByDate only indexes OPEN orders (20% of total), which can help reduce GSI storage costs
- *Item Collection Pattern* - Product table stores inventory using PK=product_id, SK=warehouse_id to eliminate separate inventory table
- *Order + OrderItems Aggregation* - Embedded as single item due to 100% access correlation
- *Strategic Denormalization* - account_rep_id duplicated in Order table for efficient queries

Finally, you can revisit the access patterns that were defined earlier. The following table shows how each access pattern is efficiently supported using the multi-table design with strategic GSIs. Each pattern uses either direct key lookups or single GSI queries, avoiding expensive scans and providing consistent performance at any scale.

S. No.	Access patterns	Query conditions
1	Look up Employee Details by Employee ID	Employee Table: GetItem(employee_id="emp_001")
2	Query Employee Details by Employee Name	EmployeeByName GSI: Query(name="John Smith")
3	Find an Employee's Phone Number(s)	Employee Table: GetItem(employee_id="emp_001")
4	Find a Customer's Phone Number(s)	Customer Table: GetItem(customer_id="cust_001")
5	Get Orders for Customer within Date Range	OrderByCustomerDate GSI: Query(customer_id="cust_001", order_date BETWEEN "2025-01-01" AND "2025-12-31")
6	Show all Open Orders within Date Range	OpenOrdersByDate GSI: Query 5 shards in parallel with multi-attribute PK (status="OPEN" + shard=0-4), SK=order_date BETWEEN "2025-01-01" AND "2025-12-31", merge results
7	See all Employees hired recently	EmployeeByHireDate GSI: Query(entity_type="EMPLOYEE", hire_date >= "2025-01-01")
8	Find all Employees in Warehouse	EmployeeByWarehouse GSI: Query(warehouse_id="wh_sea")

S. No.	Access patterns	Query conditions
9	Get all Items on Order for Product	ProductInOrders GSI: Query(product_id="prod_100")
10	Get Inventories for Product at all Warehouses	Product Table: Query(product_id="prod_100")
11	Get Customers by Account Rep	CustomerByAccountRep GSI: Query(account_rep_id="rep_001")
12	Get Orders by Account Rep	OrderByAccountRep GSI: Query(account_rep_id="rep_001")
13	Get Employees with Job Title	EmployeeByJobTitle GSI: Query(job_title="Manager")
14	Get Inventory by Product and Warehouse	Product Table: GetItem(product_id="prod_100", warehouse_id="wh_sea")
15	Get Total Product Inventory	Product Table: GetItem(product_id="prod_100", warehouse_id="prod_100")

Migrating to DynamoDB from a relational database

Migrating a relational database into DynamoDB requires careful planning to ensure a successful outcome. This guide will help you understand how this process works, what tools you have available, and then how to evaluate potential migration strategies and select one that'll fit your requirements.

Topics

- Reasons to migrate to DynamoDB
- Considerations when migrating a relational database to DynamoDB
- Understanding how a migration to DynamoDB works
- Tools to help migrate to DynamoDB
- Choosing the appropriate strategy to migrate to DynamoDB
- Performing an offline migration to DynamoDB
- Performing a hybrid migration to DynamoDB
- Performing an online migration to DynamoDB by migrating each table 1:1
- Perform an online migration to DynamoDB using a custom staging table

Reasons to migrate to DynamoDB

Migrating to Amazon DynamoDB presents a range of compelling benefits for businesses and organizations. Here are some key advantages that make DynamoDB an attractive choice for database migration:

- **Scalability**: DynamoDB is designed to handle massive workloads and scale seamlessly to accommodate growing data volumes and traffic. With DynamoDB, you can easily scale your database up or down based on demand, ensuring that your applications can handle sudden spikes in traffic without compromising performance.
- **Performance**: DynamoDB offers low-latency data access, enabling applications to retrieve and process data with exceptional speed. Its distributed architecture ensures that read and write operations are distributed across multiple nodes, delivering consistent, single-digit millisecond response times even at high request rates.
- **Fully managed**: DynamoDB is a fully managed service provided by AWS. This means that AWS handles the operational aspects of database management, including provisioning, configuration,

patching, backups, and scaling. This allows you to focus more on developing your applications and less on database administration tasks.

- **Serverless architecture**: DynamoDB supports a serverless model, known as DynamoDB on-demand, where you pay only for the actual read and write requests your application makes with no upfront capacity provisioning required. This pay-per-request model offers cost efficiency and minimal operational overhead, as you only pay for the resources you consume without the need to provision and monitor capacity.
- **NoSQL flexibility**: Unlike traditional relational databases, DynamoDB follows a NoSQL data model, providing flexibility in schema design. With DynamoDB, you can store structured, semi-structured, and unstructured data, making it well-suited for handling diverse and evolving data types. This flexibility enables faster development cycles and easier adaptation to changing business requirements.
- **High availability and durability**: DynamoDB replicates data across multiple availability zones within a Region, ensuring high availability and data durability. It automatically handles replication, failover, and recovery, minimizing the risk of data loss or service disruptions. DynamoDB provides an availability SLA of up to 99.999%.
- **Security and compliance**: DynamoDB integrates with AWS Identity and Access Management for fine-grained access control. It provides encryption at rest and in-transit, ensuring the security of your data. DynamoDB also adheres to various compliance standards, including HIPAA, PCI DSS, and GDPR, enabling you to meet regulatory requirements.
- **Integration with AWS Ecosystem**: As part of the AWS ecosystem, DynamoDB seamlessly integrates with other AWS services, such as AWS Lambda, CloudFormation, and AWS AppSync. This integration enables you to build serverless architectures, leverage infrastructure as code, and create real-time data-driven applications.

Considerations when migrating a relational database to DynamoDB

Relational database systems and NoSQL databases have different strengths and weaknesses. These differences make database design different between the two systems.

Type of task	Relational database	NoSQL database
Querying the database	In relational databases, data can be queried flexibly,	In a NoSQL database such as DynamoDB, data can be

Type of task	Relational database	NoSQL database
	but queries are relatively expensive and don't scale well in high-traffic situations (see First steps for modeling relational data in DynamoDB). A relational database application may implement business logic in stored procedures, SQL subqueries, bulk update queries, and aggregation queries.	queried efficiently in a limited number of ways, outside of which queries can be expensive and slow. Writes to DynamoDB are singletons. Application business logic that formerly ran in stored procedures must be refactored to run outside of DynamoDB in custom code running on a host such as Amazon Amazon EC2 or AWS Lambda.
Designing the database	You design for flexibility without worrying about implementation details or performance. Query optimization generally doesn't affect schema design, but normalization is important.	You design your schema specifically to make the most common and important queries as fast and as inexpensive as possible. Your data structures are tailored to the specific requirements of your business use cases.

Designing for NoSQL database requires a different mindset than designing for a relational database management system (RDBMS). For an RDBMS, you can create a normalized data model without thinking about access patterns. You can then extend it later when new questions and query requirements arise. You can organize each type of data into its own table.

With NoSQL design, you can design your schema for DynamoDB when you know the questions it'll need to answer. Understanding the business problems and the application read and write patterns is essential. You should also aim to maintain as few tables as possible in a DynamoDB application. Having fewer tables keeps things more scalable, requires less permissions management, and reduces overhead for your DynamoDB application. It can also help keep backup costs lower overall.

The task of modeling relational data for DynamoDB and building a new version of the front-end application is a separate topic. This guide assumes you have a new version of your application built to use DynamoDB, but you still need to determine how best to migrate and synchronize historical data during the cutover.

Sizing Considerations

The maximum size of each item (row) that you store in a DynamoDB table is 400KB. For more information, see the section called "Quotas". The item size is determined by the total size of all attribute names and attribute values in an item. For more information, see the section called "Item sizes and formats".

If your application needs to store more data in an item than the DynamoDB size limit permits, break the item into an item collection, compress the item data, or store the item as an object in Amazon Simple Storage Service (Amazon S3) while storing the Amazon S3 object identifier in your DynamoDB item. See the section called "Large items". The cost to update an item is based on the full size of the item. For workloads that require frequent updates to existing items, having small items of one or two KB will cost less to update than larger items. See the section called "Working with Item Collections" for more information on item collections.

When choosing the partition and sort key attributes, other table settings, item size and structure, and whether to create secondary indexes, be sure to review the DynamoDB Modeling documentation as well as the guide for the section called "Cost optimization". Be sure to test your migration plan so your DynamoDB solution is cost efficient and fits within DynamoDB's features and limitations.

Understanding how a migration to DynamoDB works

Before reviewing the migration tools available to us, consider how writes are processed by DynamoDB.

The default and most common write operation is a single `PutItem` API operation. You can perform a `PutItem` operation in a loop to process data sets. DynamoDB supports virtually unlimited concurrent connections, so assuming you can configure and run a massively multi-threaded loading routine such as MapReduce or Spark, the velocity of writes is only limited by the capacity of the target table (which is also generally unlimited).

When loading data into DynamoDB, it's important to understand your loader's write velocity. If the items (rows) you are loading are 1KB in size or less, this velocity is simply the number of items per second. The target table can then be provisioned with sufficient WCU (write capacity units) to handle this rate. If your loader exceeds the provisioned capacity in any given second, the extra requests may be throttled or rejected altogether. You can check for throttles in the CloudWatch charts found in the DynamoDB console monitoring tab.

The second operation that can be performed is with a related API called `BatchWriteItem`. `BatchWriteItem` allows you to combine up to 25 write requests into one API call. These are received by the service and processed as separate `PutItem` requests to the table. Currently, when choosing `BatchWriteItem`,you will not get the advantage of automatic retries that is included with the AWS SDK when making singleton calls with `PutItem`. So if there are any errors (such as throttling exceptions), you'll have to look for the list of any failed writes on the response call to `BatchWriteItem`. For more information on handling throttling warnings in case these are detected in the CloudWatch throttling charts, see the section called “Throttling”.

The third type of data import is possible with the DynamoDB Import from S3 feature. This feature allows you to stage a large dataset in Amazon S3 and ask DynamoDB to automatically import the data into a new table. The import is not instant and will take time proportional to the size of the dataset. However, it's convenient since it requires no ETL platform or custom DynamoDB code. DynamoDB loads the data into a new table that's created by the import. Currently, it doesn't allow you to load data into an existing table. DynamoDB imports the data as-is, with no transformations. Similar to `PutItem`, it requires an upstream process and writes the data in your chosen format to an Amazon S3 bucket.

Tools to help migrate to DynamoDB

There are several common migration and ETL tools you can use to migrate data into DynamoDB.

Amazon provides a host of data tools that can be used in migration, including AWS Database Migration Service (DMS), AWS Glue, Amazon EMR, and Amazon Managed Streaming for Apache Kafka. All of these tools can be used to perform a downtime migration, and they can leverage relational database Change Data Capture (CDC) features to support online migrations. When choosing a tool, it'll help to consider the skill set and experience your organization has with each tool along with the features, performance, and cost of each one.

Many customers choose to write their own migration scripts and jobs in order to build custom data transformations for the migration process. If you plan to operate a high volume DynamoDB

table with heavy write traffic or regular large bulk load jobs, you may wish to write migration code yourself to get more familiar with the behavior of DynamoDB under heavy write traffic. Scenarios such as throttle handling and efficient table provisioning can be experienced early in the project when performing a practice migration.

Choosing the appropriate strategy to migrate to DynamoDB

A large relational database application may span a hundred or more tables and support several different application functions. When approaching a large migration, consider breaking your application into smaller components or micro-services, and migrating a small set of tables at a time. You can then migrate additional components to DynamoDB in waves.

When selecting a migration strategy, various factors may steer you towards one solution or another. We can present these options in a decision tree to simplify the options available to us given our requirements and resources available. The concepts are briefly mentioned here (but will be covered in more depth later in the guide):

- **Offline migration**: if your application can tolerate some downtime during the migration, it'll simplify the migration process.
- **Hybrid migration**: this approach allows for partial uptime during a migration, such as allowing reads but not writes, or allowing reads and inserts but not updates and deletes.
- **Online migration**: applications that require zero downtime during migration are less easy to migrate, and can require significant planning and custom development. One key decision is to estimate and weigh the costs of building a custom migration process versus the cost to the business of having a downtime window during cutover.

If	And	Then
You're okay to take the application down for some time during a maintenance window to perform the data migration. This is an **offline migration**.		Use AWS DMS and perform an offline migration using a full load task. Pre-shape the source data with a SQL `VIEW` if desired.
You're okay to run the application in read-only mode		Disable writes within the application or source

If	And	Then
during migration. This is a **hybrid migration**.		database. Use AWS DMS and perform an offline migration using a full load task.
You're okay to run the application with reads and new record inserts, but no updates or deletes, during the migration. This is a **hybrid migration**.	You have application development skills and can update the existing relationa l app to perform dual writes including to DynamoDB, for all new records	Use AWS DMS and perform an offline migration using a full load task. Concurren tly, deploy a version of the existing app that allows reads and performs dual writes.
You require a migration with minimal downtime. This is an **online** migration.	• You're migrating source tables 1-for-1 into DynamoDB without major schema changes.	Use AWS DMS to perform an online data migration. Run a bulk load task followed by CDC sync task.
You require a migration with minimal downtime. This is an **online** migration.	• You're combining source tables into fewer DynamoDB tables following the stacked schema or single table philosophy. • You have backend database development skills and spare capacity on the SQL host.	Create the NoSQL-ready table within the SQL database. Populate and synchronize it using JOINs, UNIONs, VIEWs, triggers, stored procedures.
You require a migration with minimal downtime. This is an **online** migration.	• You're combining source tables into fewer DynamoDB tables following single table philosophy. For example: • You don't have backend database development skills and spare capacity on the SQL host.	Consider the hybrid or offline migration approaches.

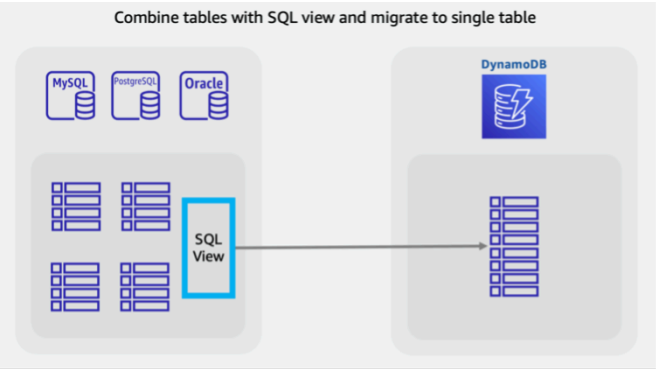
If	And	Then
You require a migration with minimal downtime. This is an **online** migration.	You're okay to skip migrating historical transaction data, or can archive it in Amazon S3 in lieu of migrating it. You just need to migrate a few small static tables.	Write a script or use any ETL tool to migrate the tables. Pre-shape the source data with a SQL `VIEW` if desired.

Performing an offline migration to DynamoDB

Offline migrations are suitable for when you can allow a downtime window to perform the migration. Relational databases commonly take at least some downtime each month for maintenance and patching, or may take longer downtimes for hardware upgrades or major release upgrades.

Amazon S3 can be used as a staging area during a migration. Data stored in CSV (comma separated values) or DynamoDB JSON format can be automatically imported into a new DynamoDB table using the DynamoDB import from S3 feature.

You may want to combine tables to leverage unique NoSQL access patterns (for example, transforming four legacy tables into a single DynamoDB table). A single key-value document request or a query for a pre-grouped item collection usually returns with better latency than a SQL database that performs a multi-table join. However, this makes the migration task more difficult. A SQL view could do the work within the source database to prepare a single dataset representing all four tables in one set.



This view can `JOIN` tables into a denormalized form, or could keep the entities normalized and stack tables using a SQL `UNION`. Key decisions around re-shaping relational data are covered in

this video. For offline migrations, using a view to combine tables is a great way to shape data for a DynamoDB single table schema.

Plan

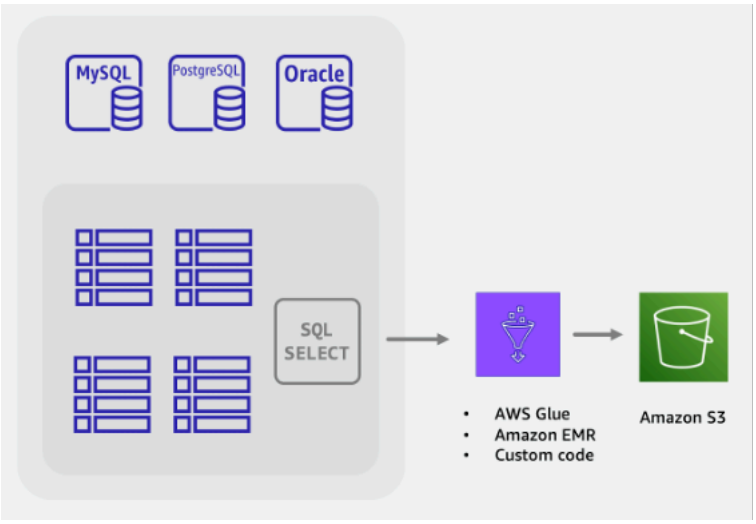
Perform an offline migration using Amazon S3

Tools

- An ETL job to extract and transform SQL data and store it in an S3 bucket such as:
  - AWS Database Migration Service, a service that can bulk load historical data and can also process Change Data Capture (CDC) records to synchronize source and target tables.
  - AWS Glue
  - Amazon EMR
  - Your own custom code
- The DynamoDB import from S3 feature

Offline migration steps:

1. Build an ETL job that can query the SQL database, transform table data into DynamoDB JSON or CSV format, and save it to an S3 bucket.



2. The DynamoDB Import from S3 feature is invoked to create a new table and automatically load data from your S3 bucket.

The fully offline migration is simple and straightforward, but it may not be popular with application owners and users. Users would benefit if the application could provide reduced levels of service during the migration, instead of no service at all.

You could add functionality to disable writes during the offline migration, while allowing reads to continue as normal. Application users could still safely browse and query for existing data while the relational data is being migrated. If this is what you're looking for, continue reading to learn about hybrid migrations.

Performing a hybrid migration to DynamoDB

While all database applications perform read and write operations, the types of write operations being performed should be considered when planning a hybrid or online migration. Database writes can be classified into three buckets: inserts, updates, and deletes. Some applications may not require immediate processing of deletions. These applications can defer deletions to a bulk cleanup process at the end of the month, for example. These types of applications can be simpler to migrate while allowing partial uptime.

Plan

Perform a hybrid online/offline migration with application dual writes

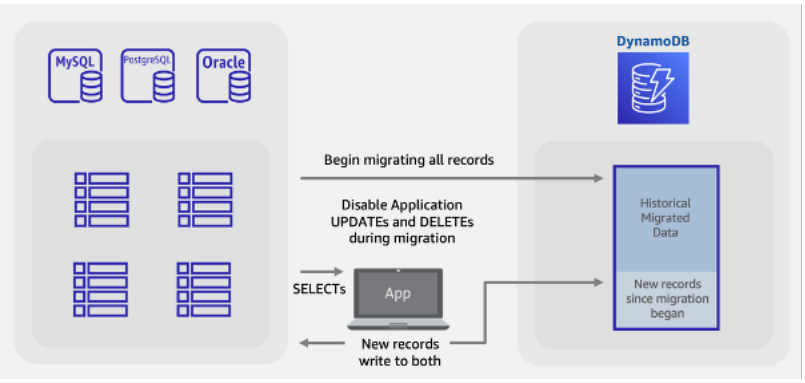
Tools

- An ETL job to extract and transform SQL data and store it in an S3 bucket such as:
  - AWS DMS
  - AWS Glue
  - Amazon EMR
  - Your own custom code

Hybrid migration steps:

1. Create the target DynamoDB table. This table will receive both historical bulk data and new, live data
2. Create a version of the legacy application that has deletes and updates disabled while performing all inserts as dual writes to both the SQL database and DynamoDB
3. Begin the ETL job or AWS DMS task to backfill existing data and deploy the new application version at the same time

4. When the backfill job completes, DynamoDB will have all existing and new records and be ready for application cutover



Note

The backfill job writes directly from SQL to DynamoDB. We are unable to use the S3 import feature as in the offline migration example, since that feature creates a new table that will not be live until after DynamoDB loads the data.

Performing an online migration to DynamoDB by migrating each table 1:1

Many relational databases have a feature called Change Data Capture (CDC), where the database allows users to request a list of the changes to a table that happened before or after a specific point in time. CDC uses internal logs to enable this feature and it does not require the table to have any timestamp column to work.

When migrating a schema of SQL tables to a NoSQL database, you might want to combine and reshape your data into fewer tables. Doing so will allow you to collect data in a single place and avoid having to manually join related data in multi-step read operations. However, single table data shaping is not always required and sometimes you'll migrate tables 1-for-1 into DynamoDB. These 1-to-1 table migrations are less complicated as you can leverage the source database CDC feature, using common ETL tools that support this type of migration. The data for each row may still be transformed into new formats, but the scope of each table remains the same.

Consider migrating SQL tables 1-to-1 into DynamoDB, with the caveat that DynamoDB doesn't support server-side joins. You'll need to add logic to your application to combine data from multiple tables.

Plan

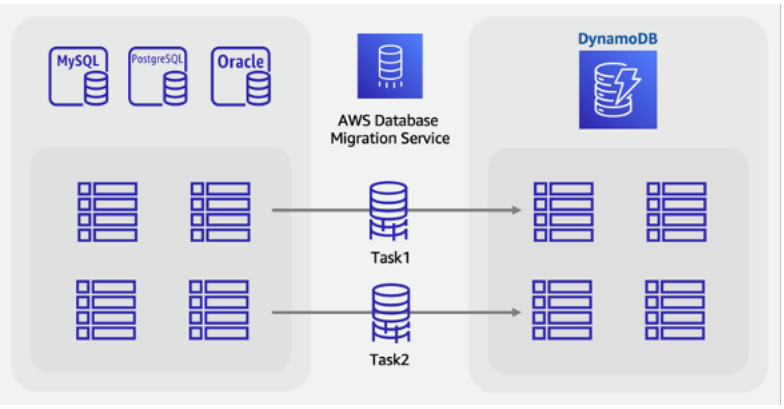
Perform an online migration of each table into DynamoDB using AWS DMS

Tools

- AWS DMS (DMS)

Online migration steps:

1. Identify the tables in your source schema that will be migrated
2. Create the same number of tables in DynamoDB with the same key structure as in the source
3. Create a replication server in AWS DMS and configure the source and target endpoints
4. Define any per-row transformations required (such as concatenated columns or conversion of dates to ISO-8601 string format)
5. Create a migration task for each table for **Full Load and Change Data Capture**
6. Monitor these tasks until the ongoing replication phase has begun
7. At this point you may perform any validation audits and then switch users to the application that reads and writes to DynamoDB



Perform an online migration to DynamoDB using a custom staging table

Like in the offline migration scenario above, you can choose to combine tables to leverage unique NoSQL access patterns (for example, transforming four legacy tables into one single DynamoDB table). A SQL `VIEW` could do the work within the source database to prepare a single dataset representing all four tables in one set.

However, for online migrations with live, changing data, you are unable to leverage CDC features as they aren't supported for `VIEW`s. If your tables include a last-updated timestamp column, and these are incorporated these into the `VIEW`, you can then build a custom ETL job that uses these to achieve a bulk load with synchronization.

A novel approach to this challenge is to use standard SQL features such as views, stored procedures, and triggers to create a new SQL table that is in the final desired DynamoDB NoSQL format.

If your database server has the spare capacity, it's possible to create this single staging table before migration begins. You would do this by writing a stored procedure that will read from existing tables, transform data as needed, and write to the new staging table. You can add a set of triggers to replicate changes in tables into the staging table in real time. If triggers are not allowed per company policy, changes to stored procedures can accomplish the same result. You would add a few lines of code to any procedure that writes data, to additionally write the same changes into the staging table.

Having this staging table in place that is fully synchronized with the legacy application tables will give you a great starting point for a live migration. Tools using database CDC to accomplish live migrations, such as AWS DMS, can now be used against this table. An advantage of this approach is that it uses well-known SQL skills and features available in the relational database engine.

Plan

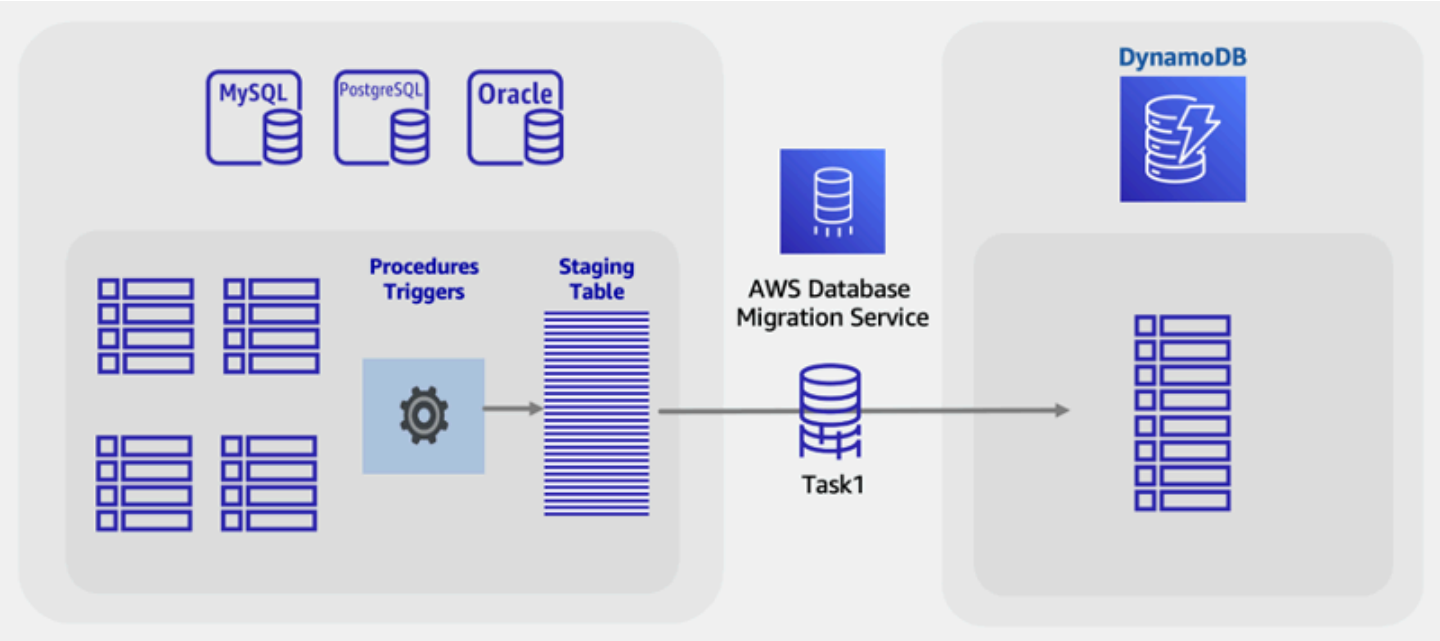
Perform an online migration with an SQL staging table using AWS DMS

Tools

- Custom SQL stored procedures or triggers
- AWS DMS

Online migration steps:

1. Within the source relational database engine, ensure there is some extra disk space and processing capacity.
2. Create a new staging table in the SQL database, with timestamps or CDC features enabled
3. Write and run a stored procedure to copy existing relational table data into the staging table
4. Deploy triggers or modify existing procedures to dual write into the new staging table while performing normal writes to existing tables
5. Run AWS DMS to migrate and synchronize this source table to a target DynamoDB table



This guide presented several considerations and approaches for migrating relational database data into DynamoDB, with a focus on minimizing downtime and using common database tools and techniques. For more information, see the following:

- AWS DMS User Guide
- AWS Glue User Guide
- Best Practices for Migrating from RDBMS to DynamoDB

NoSQL Workbench for DynamoDB

NoSQL Workbench for Amazon DynamoDB is a cross-platform, client-side GUI application that you can use for modern database development and operations. It's available for Windows, macOS, and Linux. NoSQL Workbench lets you design DynamoDB data models, define access patterns as real DynamoDB operations, and validate them using sample data. Additionally you can organize your data models into projects. NoSQL Workbench includes DynamoDB local, which makes it possible to test your tables and indexes before committing your data model to the cloud. To learn more about DynamoDB local and its requirements, see Setting up DynamoDB local (downloadable version).

Data modeler

With NoSQL Workbench for DynamoDB, you can start a new project from scratch or use a sample project that matches your use case. Then, you design tables and Global Secondary Indexes, define attributes, and configure sample data. You can also visualize your access patterns as real DynamoDB operations (PutItem, UpdateItem, Query, and others) and run these operations against the configured sample data to validate that the access pattern works as intended, making adjustments to the data model based on validation results. Finally, once validated, you commit the model to either DynamoDB local or your AWS account for further testing and production use. For collaboration, you can import and export designed data models. For more information, see Building data models with NoSQL Workbench.

Operation builder

NoSQL Workbench provides a rich graphical user interface for you to develop and test queries. You can use the *operation builder* to view, explore, and query live datasets. The structured operation builder supports projection expression, condition expression, and generates sample code in multiple languages. You can directly clone tables from one Amazon DynamoDB account to another one in different Regions. You can also directly clone tables between DynamoDB local and an Amazon DynamoDB account for faster copying of your table's key schema (and optionally GSI schema and items) between your development environments. For more information, see Exploring datasets and building operations with NoSQL Workbench.

The video below details concepts of data modeling with NoSQL Workbench.

Topics

- Download NoSQL Workbench for DynamoDB

- Building data models with NoSQL Workbench
- Exploring datasets and building operations with NoSQL Workbench
- Sample data models for NoSQL Workbench
- Release history for NoSQL Workbench

Download NoSQL Workbench for DynamoDB

Follow these instructions to download NoSQL Workbench and DynamoDB local for Amazon DynamoDB.

To download NoSQL Workbench and DynamoDB local

- Download the appropriate version of NoSQL Workbench for your operating system.

Operating system	Download link
macOS (Intel)	Download for macOS (Intel)
macOS (Apple silicon)	Download for macOS (Apple silicon)
Windows	Download for Windows
Linux	Download for Linux

Note

NoSQL Workbench includes DynamoDB local as part of the installation process. Java Runtime Environment (JRE) version 17.x or newer is required for running DynamoDB local.

Note

NoSQL Workbench supports Ubuntu 12.04, Fedora 21, and Debian 8, or any newer versions of these Linux distributions.

There are two prerequisite pieces of software required for Ubuntu installs: `libfuse2` and `curl`.
As of Ubuntu 22.04, libfuse2 is no longer installed by default. To solve this, run `sudo add-apt-repository universe && sudo apt install libfuse2` to install for the newest Ubuntu version.
For cURL, run `sudo apt update && sudo apt upgrade && sudo apt install curl`.

Building data models with NoSQL Workbench

You can use the data modeler tool in NoSQL Workbench for Amazon DynamoDB to build new data models, or to design models based on existing data models that satisfy your application data access patterns. The data modeler includes a few sample data models to help you get started.

Topics

- Creating a new data model
- Importing an existing data model
- Editing an existing data model
- Adding sample data to a data model
- Adding and validating access patterns
- Importing sample data from a CSV file
- Facets
- Viewing all tables in a data model using aggregate view
- Exporting a data model
- Committing a data model to DynamoDB

Creating a new data model

Follow these steps to create a new data model in Amazon DynamoDB using NoSQL Workbench.

To create a new data model

1. Open NoSQL Workbench, and on the main screen, select **Create model manually**.

A new page will open with an empty configuration for your first table. NoSQL Workbench creates all new data models with a default name (i.e. untitled-2) and adds them to the **Drafts** project folder.

2. On **Table configuration screen**, specify the following:

   - **Table name** — Enter a unique name for the table.
   - **Partition key** — Enter a partition key name, and specify its type. Optionally, you can also select a more granular data type format for sample data generation.
   - If you want to add a **Sort key**, specify the sort key name and its type. Optionally, you can select a more granular data type format for sample data generation.

   > **Note**
   >
   > To learn more about primary key design, designing and using partition keys effectively, and using sort keys, see the following:
   >
   > - Primary key
   > - Best practices for designing and using partition keys effectively in DynamoDB
   > - Best practices for using sort keys to organize data in DynamoDB

3. You can add other attributes to more clearly validate your model and access patterns. To add other attributes:

   - Choose **Add an attribute**.
   - Specify the attribute name and its type.
   - Optionally, you can select a more granular data type format for sample data generation.

4. If you want to add a global secondary index, choose **Add global secondary index**. Specify the **Global secondary index name**, **Partition key** attribute, and **Projection type**.

   For more information about working with global secondary indexes in DynamoDB, see Global secondary indexes.

5. Optionally, **Add a facet**. A facet is a virtual construct in NoSQL Workbench. It is not a functional construct in DynamoDB. Facets in NoSQL Workbench help you visualize an application's different data access patterns for DynamoDB with only a subset of the data in a table.

> **Note**
>
> We recommend you use Adding and validating access patterns to visualize how your application will access data in DynamoDB instead of Facets. Access patterns mirror your actual database interactions and help you build the correct data model for your use case, while facets are non-functional visualizations.

Choose **Add facet**. Specify the following:

- The **Facet name**.
- A **Partition key alias** to help distinguish this facet view.
- A **Sort key alias** if you provided a **Sort key** for the table.
- Choose the **Attributes** that are part of this facet.

Repeat this step if you want to add more facets.

6. Finally, click the **Save** button to create the table.
7. If you need other **Tables** or **Global Secondary Indexes**, click on the **+** icon above the table you just created.

Importing an existing data model

You can use NoSQL Workbench for Amazon DynamoDB to build a data model by importing and modifying an existing model. You can import data models in either NoSQL Workbench model format or in AWS CloudFormation JSON template format.

To import a data model

1. Open NoSQL Workbench, and on the main screen, select **Import model**. You can import either a NoSQL Workbench model format or CloudFormation JSON template.
2. You can add the imported model to an existing project or create a new project. If you don't choose a different project, your model will be added to the drafts project folder.
3. Click **Browse your desktop** and choose a model to import from your computer.
4. Once your model is imported, you can add access patterns, edit tables/indexes, and modify the data locally.

Editing an existing data model

To edit an existing model

1. To begin making changes to an existing model, open the model in the modeler page.
2. In the resource selector panel, you will see the list of **Tables** and **Global Secondary Indexes**.
3. To edit a **Table** or a **Global Secondary Index**, first click its name in the resource selector panel, and then use the action icons at the top. Available actions are **Delete**, **Duplicate**, and **Edit**.
4. If you want to edit **Model details**, click the three-dot icon next to the model name.
5. From there, you can click **Edit model details** and edit the information accordingly.
6. You can also **Duplicate**, **Move**, **Delete**, and **Export** the model from the same menu.
7. To change to another model, you can either go through the main screen again, or use the model selection dropdown.

Adding sample data to a data model

To auto-generate sample data

1. Open NoSQL Workbench, and on the main screen, click the name of the model that you want to add sample data to.
2. Click the additional actions (three-dot icon) in the main content toolbar and select **Add sample data**.
3. Enter the number of items of sample data that you would like to generate, then select **Confirm**.
4. Auto-generating sample data helps you generate between 1 to 5000 rows of data for immediate use. You can specify a granular sample data type to create realistic data based on your design and testing needs. To utilize the capability to generate realistic data, you need to specify the sample data type format for your attributes in the Data modeler. See Creating a new data model for specifying sample data type formats.

To add sample data one item at a time

1. Open the model you wish to edit then choose the table you want to add sample data to. Click the additional actions (three-dot icon) in the main content toolbar and select **Edit data**.

2. Now you can add, delete, and edit rows. After you have made necessary changes, click the **Save** button.

Adding and validating access patterns

You can use NoSQL Workbench for Amazon DynamoDB to create, store, and validate *access patterns*.

> **Note**
>
> See Identify your data access patterns for more details on identifying the right access patterns.

To create an access pattern

1. Open NoSQL Workbench, and on the main screen, click the name of the model that you want to add access patterns to.
2. On the left side, choose the **Access patterns** tab, and click the **+** icon.
3. On the next screen, provide a **Name**, an optional **Description**, the **Type** of the access pattern, and the **Table** or **Global Secondary Index** to test the access pattern against.

   > **Note**
   >
   > NoSQL Workbench currently supports the following operations for access patterns: `Scan`, `Query`, `GetItem`, `PutItem`, `UpdateItem`, `DeleteItem`. Amazon DynamoDB supports a broader list of operations.

4. After you create an access pattern, you can switch to the **Validate** tab to verify that your data model is designed to return expected results for the access pattern. See Adding sample data to a data model for details on how to auto-generate sample data for your tables. Different types of access patterns will support different input parameters.

   > **Note**
   >
   > To validate access patterns, NoSQL Workbench starts a separate DynamoDB local database on port `8001` (by default) with tables and indexes stored in memory.

- NoSQL Workbench automatically adds the sample data from your model to the temporary tables.
- If you edit the sample data or the data model itself, NoSQL Workbench will update the temporary tables.
- This temporary database is erased when you close the application.

To edit your access patterns

1. Open NoSQL Workbench, and on the main screen, click the name of the model that you want to edit access patterns for.
2. On the left side, choose the **Access patterns** tab.
3. To edit an access pattern, select it from the list on the left.
4. In the top bar, click the **Edit** action button.

Importing sample data from a CSV file

If you have preexisting sample data in a CSV file, you can import it into NoSQL Workbench. This enables you to quickly populate your model with sample data without having to enter it line by line.

The column names in the CSV file must match the attribute names in your data model, but they do not need to be in the same order. For example, if your data model has attributes called `LoginAlias`, `FirstName`, and `LastName`, your CSV columns could be `LastName`, `FirstName`, and `LoginAlias`.

You can import up to 150 rows at a time from a CSV file.

To import data from a CSV file into NoSQL Workbench

1. To import CSV data to a **Table**, first click the table name in the resource panel, and then click the additional actions (three-dot icon) in the main content toolbar.
2. Select **Import sample data**.
3. Select your CSV file and choose **Open**. The CSV data appends to your table.

Facets

In NoSQL Workbench, *Facets* give you a way to view a subset of the data in a table, without having to see records that don't meet the constraints of the facet. Facets are considered a visual data modeling tool, and don't exist as a usable construct in DynamoDB, as they are purely an aid to modeling of access patterns.

> **Note**
>
> We recommend you use Adding and validating access patterns to visualize how your application will access data in DynamoDB instead of Facets. Access patterns mirror your actual database interactions and help you build the correct data model for your use case, while facets are non-functional visualizations.

To create a facet

1. In the resource selector panel, choose a **Table** you wish to edit
2. In the top bar, click the **Edit** action icon.
3. Scroll down to the **Facet filters** section.
4. Choose **Add facet**. Specify the following:

   - The **Facet name**.
   - A **Partition key alias** to help distinguish this facet view.
   - A **Sort key alias** if you provided a **Sort key** for the table.
   - Choose the **Attributes** that are part of this facet.

   Repeat this step if you want to add more facets.

Viewing all tables in a data model using aggregate view

The aggregate view in NoSQL Workbench for Amazon DynamoDB allows you to visualize all the tables and indexes in a data model. For each table, the following information appears:

- Table column names
- Sample data

- All global secondary indexes that are associated with the table. NoSQL Workbench displays the following information for each index:
  - Index column names
  - Sample data

To view all table information

1. Open NoSQL Workbench, and on the main screen, click the name of the model that you want to open.
2. In the top bar, click **Aggregate view**. You will see the data across all tables and indexes on the same screen.

To export aggregate view as images

1. With **Aggregate view** selected, click the three-dot icon and choose **Export aggregate view**
2. An archive with PNG images of all tables and indexes will be presented for download.

Exporting a data model

After you create a data model using NoSQL Workbench for Amazon DynamoDB, you can save and export the model in either NoSQL Workbench model format or AWS CloudFormation JSON template format.

To export a data model

1. Open NoSQL Workbench, and on the main screen, click on the name of the model that you want to edit.
2. Click the three-dot icon next to the data model name and select **Export model**.
3. Choose whether to export your data model in NoSQL Workbench model format or CloudFormation JSON template format.
   - Choose **NoSQL Workbench model** format if you want to share your model with other team members using NoSQL Workbench or import it into NoSQL Workbench later.
   - Choose **CloudFormation JSON template** format if you want to deploy your model directly to AWS or integrate it into your infrastructure-as-code workflow.

Committing a data model to DynamoDB

When you are satisfied with your data model, you can commit the model to Amazon DynamoDB.

> **Note**
>
> - This action creates server-side resources in AWS for the tables and global secondary indexes represented in the data model.
> - NoSQL Workbench creates tables and indexes with on-demand capacity by default.

To commit the data model to DynamoDB

1. Open NoSQL Workbench, and on the main screen, click on the name of the model that you want to commit.
2. In the top bar, click **Commit**.
3. Choose an existing connection, or create a new connection by clicking the **Add new connection** button.
   - To add a new connection, specify the following information:
     - **Account Alias**
     - **AWS Region**
     - **Access key ID**
     - **Secret access key**

     For more information about how to obtain the access keys, see Getting an AWS access key.
   - You can optionally specify the following:
     - **Session token**
     - **IAM role ARN**
4. If you prefer to use DynamoDB local:
   1. Choose the **Local connection** tab.
   2. Click the **Add new connection** button.
   3. Specify the **Connection name** and **Port**.

> **Note**
>
> To use DynamoDB local, you will need to turn it on by using the **DynamoDB local** toggle at the bottom left of the NoSQL Workbench screen.

5. Click **Commit**.

Exploring datasets and building operations with NoSQL Workbench

NoSQL Workbench for Amazon DynamoDB provides a rich graphical user interface for developing and testing queries. You can use the operation builder in NoSQL Workbench to view, explore, and query live datasets. The structured operation builder supports projection expression, condition expression, and generates sample code in multiple languages. You can directly clone tables from one Amazon DynamoDB account to another one in different Regions. You can also directly clone tables between DynamoDB local and an Amazon DynamoDB account for faster copying of your table's key schema (and optionally GSI schema and items) between your development environments.You can save as many as 50 DynamoDB data operations in the operation builder.

Topics

- Connecting to live datasets
- Building complex operations
- Cloning tables with NoSQL Workbench
- Exporting data to a CSV file

Connecting to live datasets

To connect to your Amazon DynamoDB tables with NoSQL Workbench, you must first connect to your AWS account.

To add a connection to your database

1. In NoSQL Workbench, in the navigation pane on the left side, choose the **Operation builder** icon.

2. Choose **Add connection**.
3. Specify the following information:

   - **Connection name**
   - **AWS Region**
   - **Access key ID**
   - **Secret access key**

   For more information about how to obtain the access keys, see Getting an AWS access key.

   You can optionally, specify the following:

   - **Session token**
   - **IAM role ARN**
4. Choose **Connect**.

   If you don't want to sign up for a free tier account, and prefer to use DynamoDB local (downloadable version):

   a. Choose the **Local** tab on the connection screen.

   b. Specify the following information:

      - **Connection name**
      - **Port**

   c. Choose the **connect** button.

   > **Note**
   >
   > To connect to DynamoDB local, either manually launch DynamoDB local using your terminal (see deploying DynamoDB local on your computer) or launch DynamoDB local directly using the DDB local toggle in the NoSQL Workbench navigation menu. Ensure the connection port is the same as your DynamoDB local port.

5. On the created connection, choose **Open**.

After connecting to your DynamoDB database, the list of available tables appears in the left pane. Choose one of the tables to return a sample of the data stored in the table.

You can now run queries against the selected table.

To run queries on a table, see the next section on building operations see Building complex operations.

Building complex operations

The operation builder in NoSQL Workbench for Amazon DynamoDB provides a visual interface where you can perform complex data plane operations. It includes support for projection expressions and condition expressions. Once you've built an operation, you can save it for later use (up to 50 operations can be saved). You can then browse a list of your frequently used data-plane operations in the **Saved Operations** menu, and use them to automatically populate and build a new operation. You can also generate sample code for these operations, in multiple languages.

NoSQL Workbench supports building PartiQL for DynamoDB statements, which allows you to interact with DynamoDB using a SQL-compatible query language. NoSQL Workbench also supports building DynamoDB CRUD API operations.

To use NoSQL Workbench to build operations, in the navigation pane on the left side, choose the **Operation builder** icon.

Topics

- Building PartiQL statements
- Building API operations

Building PartiQL statements

To use NoSQL Workbench to build PartiQL for DynamoDB statements, choose **PartiQL editor** near the top of the NoSQL Workbench UI.

You can build the following PartiQL statement types in the operation builder.

Topics

- Singleton statements
- Transactions
- Batch

Singleton statements

To run or generate code for a PartiQL statement, do the following.

1. Choose **PartiQL editor** near the top of the window.
2. Enter a valid PartiQL statement.
3. If your statement uses parameters:
   a. Choose **Optional request parameters**.
   b. Choose **Add new parameters**.
   c. Enter the attribute type and value.
   d. If you want to add additional parameters, repeat steps b and c.
4. If you want to generate code, choose **Generate code**.

   Select your desired language from the displayed tabs. You can now copy this code and use it in your application.
5. If you want the operation to be run immediately, choose **Run**.
6. If you want to save this operation for later use, choose **Save operation**. Then enter a name for your operation and choose **Save**.

Transactions

To run or generate code for a PartiQL transaction, do the following.

1. Choose **PartiQLTransaction** from the **More operations** dropdown.
2. Choose **Add a new statement**.
3. Enter a valid PartiQL statement.

> **Note**
>
> Read and write operations are not supported in the same PartiQL transaction request. A SELECT statement cannot be in the same request with INSERT, UPDATE, and DELETE statements. See Performing transactions with PartiQL for DynamoDB for more details.

4. If your statement uses parameters
   a. Choose **Optional request parameters**.

b. Choose **Add new parameters**.

c. Enter the attribute type and value.

d. If you want to add additional parameters, repeat steps b and c.

5. If you want to add more statements, repeat steps 2 to 4.
6. If you want to generate code, choose **Generate code**.

   Select your desired language from the displayed tabs. You can now copy this code and use it in your application.
7. If you want the operation to be run immediately, choose **Run**.
8. If you want to save this operation for later use, choose **Save operation**. Then enter a name for your operation and choose **Save**.

Batch

To run or generate code for a PartiQL batch, do the following.

1. Choose **PartiQLBatch** from the **More operations** dropdown.
2. Choose **Add a new statement**.
3. Enter a valid PartiQL statement.

> **Note**
>
> Read and write operations are not supported in the same PartiQL batch request, which means a SELECT statement cannot be in the same request with INSERT, UPDATE, and DELETE statements. Write operations to the same item are not allowed. As with the BatchGetItem operation, only singleton read operations are supported. Scan and query operations are not supported. See Running batch operations with PartiQL for DynamoDB for more details.

4. If your statement uses parameters:

   a. Choose **Optional request parameters**.

   b. Choose **Add new parameters**.

   c. Enter the attribute type and value.

   d. If you want to add additional parameters, repeat steps b and c.

5. If you want to add more statements, repeat steps 2 to 4.
6. If you want to generate code, choose **Generate code**.

   Select your desired language from the displayed tabs. You can now copy this code and use it in your application.
7. If you want the operation to be run immediately, choose **Run**.
8. If you want to save this operation for later use, choose **Save operation**. Then enter a name for your operation and choose **Save**.

Building API operations

To use NoSQL Workbench to build DynamoDB CRUD APIs, select **Operation builder** from the left of the NoSQL Workbench user interface.

Then select **Open** and choose a connection.

You can perform the following operations in the operation builder.

- Delete Table
- Create Table
- Update Table

- Put Item
- Update Item
- Delete Item
- Query
- Scan
- Transact Get Items
- Transact Write Items

Delete table

To run a `Delete Table` operation, do the following.

1. Find the table you want to delete from the **Tables** section.

2. Select **Delete Table** from the horizontal ellipsis menu.
3. Confirm you want to delete the table by entering the **Table name**.
4. Select **Delete**.

For more information about this operation, see Delete table in the *Amazon DynamoDB API Reference*.

Delete GSI

To run a `Delete GSI` operation, do the following.

1. Find the GSI of a table you want to delete from the **Tables** section.
2. Select **Delete GSI** from the horizontal ellipsis menu.
3. Confirm you want to delete the GSI by entering the **GSI name**.
4. Select **Delete**.

For more information about this operation, see Delete table in the *Amazon DynamoDB API Reference*.

Create table

To run a `Create Table` operation, do the following.

1. Choose the **+** icon next to the **Tables** section.
2. Enter the table name desired.
3. Create a partition key.
4. Optional: create a sort key.
5. To customize capacity settings, and uncheck the box next to **Use default capacity settings**.
   - You can now select either **Provisioned** or **On-demand capacity.**

     With Provisioned selected, you can set minimum and maximum read and write capacity units. You can also enable or disable auto scaling.
   - If the table is currently set to On-demand, you will be unable to specify a provisioned throughput.
   - If you switch from On-demand to Provisioned throughput, then Autoscaling will automatically be applied to all GSIs with: min: 1, max: 10; target: 70%.

6. Select **Skip GSIs and create** to create this table without a GSI. Optionally, you can select **Next** to create a GSI with this new table.

For more information about this operation, see Create table in the *Amazon DynamoDB API Reference*.

Create GSI

To run a `Create GSI` operation, do the following.

1. Find a table that you want to add a GSI to.
2. From the horizontal ellipsis menu, select **Create GSI**.
3. Name your GSI under **Index name**.
4. Create a partition key.
5. Optional: create a sort key.
6. Choose a projection type option from the dropdown.
7. Select **Create GSI**.

For more information about this operation, see Create table in the *Amazon DynamoDB API Reference*.

Update table

To update capacity settings for a table with an `Update Table` operation, do the following.

1. Find the table you want to update capacity settings for.
2. From the horizontal ellipsis menu, select **Update capacity settings**.
3. Select either **Provisioned** or **On-demand capacity.**

   With **Provisioned** selected, you can set minimum and maximum read and write capacity units. You can also enable or disable auto scaling.
4. Select **Update**.

For more information about this operation, see Update table in the *Amazon DynamoDB API Reference*.

Update GSI

To update capacity settings for a GSI with an `Update Table` operation, do the following.

> **Note**
>
> By default, global secondary indexes inherit the capacity settings of the base table. Global secondary indexes can have a different capacity mode only when the base table is in provisioned capacity mode. When you create a global secondary index on a provisioned mode table, you must specify read and write capacity units for the expected workload on that index. For more information, see Provisioned throughput considerations for Global Secondary Indexes.

1. Find the GSI you want to update capacity settings for.
2. From the horizontal ellipsis menu, select **Update capacity settings**.
3. You can now select either **Provisioned** or **On-demand capacity.**

   With **Provisioned** selected, you can set minimum and maximum read and write capacity units. You can also enable or disable auto scaling.
4. Select **Update**.

For more information about this operation, see Update table in the *Amazon DynamoDB API Reference*.

Put item

You create an item by using the `Put Item` operation. To run or generate code for a `Put Item` operation, do the following.

1. Find the table you want to create an item in.
2. From the **Actions** dropdown, select **Create item**.
3. Enter the partition key value.
4. Enter the sort key value, if one exists.
5. If you want to add non-key attributes, do the following:

   a. Select **+ Add other attributes**.

   b. Specify the **Attribute name**, **Type**, and **Value**.

6. If a condition expression must be satisfied for the `Put Item` operation to succeed, do the following:

   a. Choose **Condition**.

   b. Specify the attribute name, comparison operator, attribute type, and attribute value.

   c. If other conditions are needed, choose **Condition** again.

   For more information, see DynamoDB condition expression CLI example.

7. If you want to generate code, choose **Generate code**.

   Select your desired language from the displayed tabs. You can now copy this code and use it in your application.

8. If you want the operation to be run immediately, choose **Run**.

9. If you want to save this operation for later use, choose **Save operation**, then enter a name for your operation and choose **Save**.

For more information about this operation, see PutItem in the *Amazon DynamoDB API Reference*.

Update item

To run or generate code for an `Update Item` operation, do the following:

1. Find the table you want to update an item in.
2. Select the item.
3. Enter the attribute name and attribute value for the selected expression.
4. If you want to add more expressions, choose another expression in the **Update Expression** dropdown list, and then select the **+** icon.
5. If a condition expression must be satisfied for the `Update Item` operation to succeed, do the following:

   a. Choose **Condition**.

   b. Specify the attribute name, comparison operator, attribute type, and attribute value.

   c. If other conditions are needed, choose **Condition** again.

   For more information, see DynamoDB condition expression CLI example.

6. If you want to generate code, choose **Generate code**.

   Choose the tab for the language that you want. You can now copy this code and use it in your application.
7. If you want the operation to be run immediately, choose **Run**.
8. If you want to save this operation for later use, choose **Save operation**, then enter a name for your operation and choose **Save**.

For more information about this operation, see UpdateItem in the *Amazon DynamoDB API Reference*.

Delete item

To run a `Delete Item` operation, do the following.

1. Find the table you want to delete an item in.
2. Select the item.
3. From the **Actions** dropdown, select **Delete item**.
4. Confirm you want to delete the item by selecting **Delete**.

For more information about this operation, see DeleteItem in the *Amazon DynamoDB API Reference*.

Duplicate item

You can duplicate an item by creating a new item with the same attributes. To duplicate an item, do the following.

1. Find the table you want to duplicate an item in.
2. Select the item.
3. From the **Actions** dropdown, select **Duplicate item**.
4. Specify a new partition key.
5. Specify a new sort key (if necessary).
6. Select **Run**.

For more information about this operation, see DeleteItem in the *Amazon DynamoDB API Reference*.

Query

To run or generate code for a `Query` operation, do the following.

1. Select **Query** from the top of the NoSQL Workbench UI.
2. Specify the partition key value.
3. If a sort key is needed for the `Query` operation:
   a. Select **Sort key**.
   b. Specify the comparison operator, and attribute value.
4. Select **Query** to run this query operation. If more options are needed, check the **More options** checkbox and continue on with the following steps.
5. If not all the attributes should be returned with the operation result, select **Projection expression**.
6. Choose the **+** icon.
7. Enter the attribute to return with the query result.
8. If more attributes are needed, choose the **+** .
9. If a condition expression must be satisfied for the `Query` operation to succeed, do the following:
   a. Choose **Condition**.
   b. Specify the attribute name, comparison operator, attribute type, and attribute value.
   c. If other conditions are needed, choose **Condition** again.

   For more information, see DynamoDB condition expression CLI example.
10. If you want to generate code, choose **Generate code**.

    Choose the tab for the language that you want. You can now copy this code and use it in your application.
11. If you want the operation to be run immediately, choose **Run**.
12. If you want to save this operation for later use, choose **Save operation**, then enter a name for your operation and choose **Save**.

For more information about this operation, see Query in the *Amazon DynamoDB API Reference*.

Scan

To run or generate code for a `Scan` operation, do the following.

1. Select **Scan** from the top of the NoSQL Workbench UI.
2. Select the **Scan** button to perform this basic scan operation. If more options are needed, check the **More options** checkbox and continue on with the following steps.
3. Specify an attribute name to filter your scan results.
4. If not all the attributes should be returned with the operation result, select **Projection expression**.
5. If a condition expression must be satisfied for the scan operation to succeed, do the following:

   a. Choose **Condition**.

   b. Specify the attribute name, comparison operator, attribute type, and attribute value.

   c. If other conditions are needed, choose **Condition** again.

   For more information, see DynamoDB condition expression CLI example.
6. If you want to generate code, choose **Generate code**.

   Choose the tab for the language that you want. You can now copy this code and use it in your application.
7. If you want the operation to be run immediately, choose **Run**.
8. If you want to save this operation for later use, choose **Save operation**, then enter a name for your operation and choose **Save**.

TransactGetItems

To run or generate code for a `TransactGetItems` operation, do the following.

1. From the **More operations** dropdown at the top of the NoSQL Workbench UI, choose **TransactGetItems**.
2. Choose the **+** icon near **TransactGetItem**.
3. Specify a partition key.
4. Specify a sort key (if necessary).

5. Select **Run** to perform the operation, **Save operation** to save it, or **Generate code** to generate code for it.

For more information about transactions, see Amazon DynamoDB transactions.

TransactWriteItems

To run or generate code for a `TransactWriteItems` operation, do the following.

1. From the **More operations** dropdown at the top of the NoSQL Workbench UI, choose **TransactWriteItems**.
2. Choose an operation from the **Actions** dropdown.
3. Choose the **+** icon near **TransactWriteItem**.
4. In the **Actions** dropdown, choose the operation that you want to perform.

   - For `DeleteItem`, follow the instructions for the Delete item operation.
   - For `PutItem`, follow the instructions for the Put item operation.
   - For `UpdateItem`, follow the instructions for the Update item operation.

   To change the order of actions, choose an action in the list on the left side, and then choose the up or down arrows to move it up or down in the list.

   To delete an action, choose the action in the list, and then choose the **Delete** (trash can) icon.
5. Select **Run** to perform the operation, **Save operation** to save it, or **Generate code** to generate code for it.

For more information about transactions, see Amazon DynamoDB transactions.

Cloning tables with NoSQL Workbench

Cloning tables will copy a table's key schema (and optionally GSI schema and items) between your development environments. You can clone a table between DynamoDB local to an Amazon DynamoDB account, and even clone a table from one account to another in different Regions for faster experimentation.

To clone a table

1. In the **Operation Builder**, select your connection and Region (Region selection is not available for DynamoDB local).
2. Once you are connected to DynamoDB, browse your tables and select the table you want to clone.
3. From the horizontal ellipsis menu, select the **Clone** option.
4. Input your clone destination details:
    a. Select a connection.
    b. Select a Region (Region is not available for DynamoDB local).
    c. Enter a new table name.
    d. Choose a clone option:
        i. **Key schema** is selected by default and cannot be unselected. By default, cloning a table will copy your primary key and sort key if they are available.
        ii. **GSI schema** is selected by default if your table to be cloned has a GSI. Cloning a table will copy your GSI primary key and sort key if they are available. You have the option to deselect GSI schema to skip cloning the GSI schema. Cloning a table will copy your base table's capacity settings as the GSI's capacity settings. You can use the `UpdateTable` operation in Operation Builder to update the table's GSI capacity setting after cloning is complete.
5. Enter the number of items to clone. To only clone the key schema and optionally the GSI schema, you can keep the **Items to clone** value at 0. The maximum number of items that can be cloned is 5000.
6. Choose a capacity mode:
    a. **On-demand mode** is selected by default. DynamoDB on-demand offers pay-per-request pricing for read and write requests so that you pay only for what you use. To learn more, see DynamoDB On-demand mode .
    b. **Provisioned mode** lets you specify the number of reads and writes per second that you require for your application. You can use auto scaling to adjust your table's provisioned capacity automatically in response to traffic changes. To learn more, see DynamoDB Provisioned mode.
7. Select **Clone** to begin cloning.

8. The cloning process will run in the background. The **Operation builder** tab will show a notification when there is a change in the cloning table status. You can access this status by selecting the **Operation builder** tab and then selecting the arrow button. The arrow button is located on the cloning table status widget located near the bottom of the menu sidebar.

Exporting data to a CSV file

You can export the results of a query from Operation Builder to a CSV file. This enables you to load the data into a spreadsheet or process it using your preferred programming language.

Exporting to CSV

1. In the Operation Builder, run an operation of your choice, such as a Scan or Query.

   > **Note**
   >
   > - You can only export results from read API operations and PartiQL statements to a CSV file. You can't export results from transaction read statements.
   > - Currently, you can export results one page at a time to a CSV file. If there are multiple pages of results, you must export each page individually.

2. Select the items you want to export from the results.
3. In the **Actions** dropdown, choose **Export as CSV**.
4. Choose a filename and location for your CSV file and select **Save**.

Sample data models for NoSQL Workbench

The home page for the modeler displays a number of sample models that ship with the NoSQL Workbench. This section describes these models and their potential uses.

Topics

- Employee data model
- Discussion forum data model
- Music library data model
- Ski resort data model

- Credit card offers data model
- Bookmarks data model

Employee data model

This data model is an introductory model. It represents an employee's basic details such as a unique alias, first name, last name, designation, manager, and skills.

This data model depicts a few techniques such as handling complex attribute such as having more than one skill. This model is also an example of one-to-many relationship through the manager and their reporting employees that has been achieved by the secondary index DirectReports.

The access patterns facilitated by this data model are:

- Retrieval of an employee record using the employee's login alias, facilitated by a table called `Employee`.
- Search for employees by name, facilitated by the Employee table's global secondary index called `Name`.
- Retrieval of all direct reports of a manager using the manager's login alias, facilitated by the Employee table's global secondary index called `DirectReports`.

Discussion forum data model

This data model represents a discussion forums. Using this model customers can engage with the developer community, ask questions, and respond to other customers' posts. Each AWS service has a dedicated forum. Anyone can start a new discussion thread by posting a message in a forum, and each thread receives any number of replies.

The access patterns facilitated by this data model are:

- Retrieval of a forum record using the forum's name, facilitated by a table called `Forum`.
- Retrieval of a specific thread or all threads for a forum, facilitated by a table called `Thread`.
- Search for replies using the posting user's email address, facilitated by the Reply table's global secondary index called `PostedBy-Message-Index`.

Music library data model

This data model represents a music library that has a large collection of songs and showcases its most downloaded songs in near-real time.

The access patterns facilitated by this data model are:

- Retrieval of a song record, facilitated by a table called `Songs`.
- Retrieval of a specific download record or all download records for a song, facilitated by a table called `Songs`.
- Retrieval of a specific monthly download count record or all monthly download count records for a song, facilitated by a table called Song.
- Retrieval of all records (including song record, download records, and monthly download count records) for a song, facilitated by a table called `Songs`.
- Search for most downloaded songs, facilitated by the Songs table's global secondary index called `DownloadsByMonth`.

Ski resort data model

This data model represents a ski resort that has a large collection of data for each ski lift collected daily.

The access patterns facilitated by this data model are:

- Retrieval of all data for a given ski lift or overall resort, dynamic and static, facilitated by a table called `SkiLifts`.
- Retrieval of all dynamic data (including unique lift riders, snow coverage, avalanche danger, and lift status) for a ski lift or the overall resort on a specific date, facilitated by a table called `SkiLifts`.
- Retrieval of all static data (including if the lift is for experienced riders only, vertical feet the lift rises, and lift riding time) for a specific ski lift, facilitated by a table called `SkiLifts`.
- Retrieval of date of data recorded for a specific ski lift or the overall resort sorted by total unique riders, facilitated by the SkiLifts table's global secondary index called `SkiLiftsByRiders`.

Credit card offers data model

This data model is used by a Credit Card Offers Application.

A credit card provider produces offers over time. These offers include balance transfers without fees, increased credit limits, lower interest rates, cash back, and airline miles. After a customer accepts or declines these offers, the respective offer status is updated accordingly.

The access patterns facilitated by this data model are:

- Retrieval of account records using `AccountId`, as facilitated by the main table.
- Retrieval of all the accounts with few projected items, as facilitated by the secondary index `AccountIndex`.
- Retrieval of accounts and all the offer records associated with those accounts by using `AccountId`, as facilitated by the main table.
- Retrieval of accounts and specific offer records associated with those accounts by using `AccountId` and `OfferId`, as facilitated by the main table.
- Retrieval of all `ACCEPTED/DECLINED` offer records of specific `OfferType` associated with accounts using `AccountId`, `OfferType`, and `Status`, as facilitated by the secondary index `GSI1`.
- Retrieval of offers and associated offer item records using `OfferId`, as facilitated by the main table.

Bookmarks data model

This data model is used store bookmarks for customers.

A customer can have many bookmarks and a bookmark can belong to many customers. This data model represents a many-to-many relationship.

The access patterns facilitated by this data model are:

- A single query by `customerId` can now return customer data as well as bookmarks.
- A query `ByEmail` index returns customer data by email address. Note that bookmarks are not retrieved by this index.
- A query `ByUrl` index gets bookmarks data by URL. Note that we have customerId as the sort key for the index because the same URL can be bookmarked by multiple customers.

- A query `ByCustomerFolder` index gets bookmarks by folder for each customer.

Release history for NoSQL Workbench

The following table describes the important changes in each release of the *NoSQL Workbench* client tool.

Version	Change	Description	Date
3.20.0	New Data Modeler for DynamoDB	Data Modeler for DynamoDB has an updated user experience. Data Modeler for DynamoDB now supports access patterns.	February 16, 2026
3.13.5	Capacity mode for default table settings is now on-demand	When you create a table with default settings, DynamoDB creates a table that uses on-demand capacity mode instead of provision ed capacity mode.	February 24, 2025
3.13.0	NoSQL Workbench operation builder improvements	NoSQL Workbench now includes native support for dark mode. Improved table and item operations in the operations builder. Item results and operation builder request information	April 24, 2024

Version	Change	Description	Date
		is available in JSON format.	
3.12.0	Cloning tables with NoSQL Workbench and returning capacity consumed	You can now clone tables between DynamoDB local and a DynamoDB web service account or between DynamoDB web service accounts for faster developme nt iterations. View RCU or WCU consumed after running an operation using the Operations Builder. We fixed the overwrite data issue when importing from a CSV file.	February 26, 2024
3.11.0	DynamoDB local improvements	You can now specify port when launching the built-in DynamoDB local instance. NoSQL Workbench can now be installed on Windows without admin rights. We have updated the data model templates.	January 17, 2024

Version	Change	Description	Date
3.10.0	Native support for Apple silicon	NoSQL Workbench now includes native support for Mac with Apple silicon. You can now configure sample data generation format for attributes of type `Number`.	December 5, 2023
3.9.0	Data modeler improvements	Visualizer now supports committin g data models to DynamoDB local with the option to overwrite existing tables.	November 3, 2023
3.8.0	Sample data generation	NoSQL Workbench now supports generating sample data for your DynamoDB data models.	September 25, 2023
3.6.0	Improvements in the Operations builder	Connections management improvements in the Operations builder. Attribute names in Data Modeler can now be changed without deleting data. Other bug fixes.	April 11, 2023

Version	Change	Description	Date
3.5.0	Support for new AWS Regions	NoSQL Workbench now supports the ap-south-2, ap-southeast-3, ap-southeast-4, eu-central-2, eu-south-2, me-central-1, and me-west-1 regions.	February 23, 2023
3.4.0	Support for DynamoDB local	NoSQL Workbench now supports installing DynamoDB local as part of the installation process.	December 6, 2022
3.3.0	Support for control plane operations	Operation Builder now supports control plane operations.	June 1, 2022
3.2.0	CSV import and export	You can now import sample data from a CSV file in the Visualizer tool, and also export the results of an Operation Builder query to a CSV file.	October 11, 2021
3.1.0	Save operations	The Operation Builder in NoSQL Workbench now supports saving operations for later use.	July 12, 2021

Version	Change	Description	Date
3.0.0	Capacity settings and CloudFormation import/export	NoSQL Workbench for Amazon DynamoDB now supports specifying a read/write capacity mode for tables, and can now import and export data models in CloudFormation format.	April 21, 2021
2.2.0	Support for PartiQL	NoSQL Workbench for Amazon DynamoDB adds support for building PartiQL statements for DynamoDB.	December 4, 2020
1.1.0	Support for Linux.	NoSQL Workbench for Amazon DynamoDB is supported on Linux—Ubuntu, Fedora, and Debian.	May 4, 2020
1.0.0	NoSQL Workbench for Amazon DynamoDB – GA.	NoSQL Workbench for Amazon DynamoDB is generally available.	March 2, 2020

Version	Change	Description	Date
0.4.1	Support for IAM roles and temporary security credentials.	NoSQL Workbench for Amazon DynamoDB adds support for AWS Identity and Access Management (IAM) roles and temporary security credentials.	December 19, 2019
0.3.1	Support for DynamoDB local (Downloadable Version).	The NoSQL Workbench now supports connecting to DynamoDB local (Downloadable Version) to design, create, query, and manage DynamoDB tables.	November 8, 2019
0.2.1	NoSQL Workbench preview released.	This is the initial release of NoSQL Workbench for DynamoDB. Use NoSQL Workbench to design, create, query, and manage DynamoDB tables.	September 16, 2019

Backup and restore for DynamoDB

DynamoDB offers on-demand backups and point-in-time recovery (PITR) backups to help protect your DynamoDB data from disaster events and offers data archiving for long-term retention. You can back up tables from a few megabytes to hundreds of terabytes of data, with no impact on performance and availability to your production applications. All backups are automatically encrypted, cataloged, and easily discoverable.

With on-demand backups, you can create a snapshot backup of your table that DynamoDB stores and manages. You're charged based on the size and duration of your backups. Using on-demand backup, you can restore your entire DynamoDB table to the exact state it was in when the backup was created.

There are two options for creating and managing DynamoDB on-demand backups:

- DynamoDB
- AWS Backup

You can use the DynamoDB on-demand backup capability to create full backups of your tables for long-term retention, and archiving for regulatory compliance needs. You can back up and restore your table data anytime from the AWS Management Console or with a single API call.

Point-in-time recovery (PITR) backups are fully managed by DynamoDB and provide up to 35 days of recovery points at a per second granularity. To use point-in-time recovery, which are continuous backups, enable point-in-time recovery (PITR) on your DynamoDB table. You get charged based on the size of your DynamoDB table for the duration you have PITR enabled on the table. Enabling Point-in-Time Recovery (PITR) on your DynamoDB table continuously backs up your data. This helps you restore your table to a specific point in time within the PITR recovery period by creating a new DynamoDB table with the exact state of your original table at that point in time.

Point-in-time recovery helps protect your DynamoDB tables from accidental write or delete operations. With point-in-time recovery, you don't have to worry about creating, maintaining, or scheduling on-demand backups. For example, suppose that a test script writes accidentally to a production DynamoDB table.

With point-in-time recovery, you can restore your table to any point in time during the last 35 days. You can set the recovery period to any value between 1 and 35 days. After you enable point-in-

time recovery, you can restore to any point in time from five minutes before the current time until the configured recovery period. DynamoDB maintains incremental backups of your table.

In addition, point-in-time operations don't affect performance or API latencies.

You can restore a DynamoDB table to a point in time using the AWS Management Console, the AWS Command Line Interface (AWS CLI), or the DynamoDB API. The point-in-time recovery process restores to a new table.

For more information about AWS Region availability and pricing, see Amazon DynamoDB pricing.

Note

- Tagging and attribute-based access control (ABAC) aren't supported for DynamoDB backups. To use ABAC with backups, we recommend that you use AWS Backup.
- Tags aren't preserved in restored tables. You need to add tags to restored tables before you can use tag-based conditions in your policies.

The following video will give you an introductory look at the backup and restore concept and talk more about point-in-time recovery.

Backup and restore

Topics

- Point-in-time backups for DynamoDB
- Using on-demand DynamoDB backup and restore
- Understanding Amazon DynamoDB billing for backups
- Restore a table in DynamoDB
- Using AWS Backup with DynamoDB

Point-in-time backups for DynamoDB

Amazon DynamoDB point-in-time recovery (PITR) provides automatic continuous backups of your DynamoDB table data. Point-in-time recovery (PITR) backups are fully managed by DynamoDB and provide up to 35 days of recovery points at a per second granularity. With point-in-time recovery,

you don't have to worry about creating, maintaining, or scheduling on-demand backups. This section provides an overview of how the process works in DynamoDB.

Before you begin

Before you enable point-in-time recovery (PITR) on an Amazon DynamoDB table, consider the following:

- Setting the `RecoveryPeriodinDays` allows you to shorten the time period for which continuous backups are taken. By default, your `RecoveryPeriodinDays` is 35. However, you can set it to be any value between 1 and 35. Shortening the `RecoveryPeriodinDays` has no impact on PITR pricing because the price is based on the size of table and local secondary indexes.
- If you disable point-in-time recovery and later re-enable it on a table, you reset the start time for which you can recover that table. As a result, you can only immediately restore that table using the `LatestRestorableDateTime`.
- You can enable point-in-time recovery on each local replica of a global table. When you restore the table, the backup restores to an independent table that is not part of the global table. If you are using Global Tables version 2019.11.21 (Current) of global tables, you can create a new global table from the restored table. For more information, see How DynamoDB global tables work.
- You can also restore your DynamoDB table data across AWS Regions such that the restored table is created in a different Region from where the source table resides. You can do cross-Region restores between AWS commercial Regions, AWS China Regions, and AWS GovCloud (US) Regions. You pay only for the data you transfer out of the source Region and for restoring to a new table in the destination Region.
- AWS CloudTrail logs all console and API actions for point-in-time recovery to enable logging, continuous monitoring, and auditing. For more information, see Logging DynamoDB operations by using AWS CloudTrail.

Topics

- Enable point-in-time recovery in DynamoDB

Enable point-in-time recovery in DynamoDB

Amazon DynamoDB point-in-time recovery (PITR) provides automatic backups of your DynamoDB table data. This section provides an overview of how the process works in DynamoDB.

> **Note**
>
> DynamoDB charges for PITR based on the size of each DynamoDB table, including table data and local secondary indexes. Changing the recovery window (for example, from 35 days to 1 day) doesn't reduce the price. The cost remains the same regardless of the recovery period you choose. The configured maximum recovery period doesn't impact the price you're charged for turning on PITR. To determine your backup charges, DynamoDB continuously monitors the size of the tables that have PITR turned on. You're billed for PITR usage until you turn off PITR for each table.

Topics

- Enabling point-in-time recovery
- Enable PITR (console)
- Enable PITR (AWS CLI)
- Enable PITR (CloudFormation)
- Enable PITR (API)
- Recovery Period
- Edit PITR
- Delete a table with PITR enabled

Enabling point-in-time recovery

You can enable point-in-time recovery using the AWS Management Console, AWS Command Line Interface (AWS CLI), or the DynamoDB API. When enabled, point-in-time recovery provides continuous backups until you explicitly turn it off.

After you enable point-in-time recovery, you can restore to any point in time within `EarliestRestorableDateTime` and `LatestRestorableDateTime`. `LatestRestorableDateTime` is typically five minutes before the current time. For more information, see Restoring a DynamoDB table to a point in time.

> **Note**
>
> The point-in-time recovery process always restores to a new table.

Enable PITR (console)

To enable PITR using the DynamoDB console

1. Navigate to the DynamoDB console.
2. Choose **Tables** from the left navigation and select your DynamoDB table.
3. From the **Backups** tab, for the **Point in Time Recovery** option, choose **Edit**.
4. Choose **Turn on point-in-time recovery**.
5. Choose a value between 1 and 35 for your backup recovery period. This indicates the maximum time period for which the continuous backup is recoverable.

Enable PITR (AWS CLI)

> **Note**
>
> If you receive errors when running AWS CLI commands, see Troubleshoot AWS CLI errors. Make sure you're using the most recent AWS CLI version.

Run the update-continuous-backups command with the **point-in-time-recovery-specification** setting turned on:

```
aws dynamodb update-continuous-backups \
--table-name Music \
--point-in-time-recovery-specification
 PointInTimeRecoveryEnabled=true,RecoveryPeriodInDays=35
```

Enable PITR (CloudFormation)

Use the AWS::DynamoDB::Table resource with the `PointInTimeRecoverySpecification` property turned on:

```
Resources:
```

```
  iotCatalog:
    Type: AWS::DynamoDB::Table
      Properties:
      ...
      PointInTimeRecoverySpecification:
        PointInTimeRecoveryEnabled: true
        RecoveryPeriodInDays: 35
```

Request syntax example:

```
{
   "PointInTimeRecoverySpecification": {
      "PointInTimeRecoveryEnabled": boolean,
      "RecoveryPeriodInDays: number
   },
   "TableName": "string"
}
```

Enable PITR (API)

Run the UpdateContinuousBackups API operation with the `PointInTimeRecoverySpecification` parameter turned on.

Request syntax example:

```
{
   "PointInTimeRecoverySpecification": {
      "PointInTimeRecoveryEnabled": boolean,
      "RecoveryPeriodInDays" : number
   },
   "TableName": "string"
}
```

Response syntax example:

```
{
   "ContinuousBackupsDescription": {
      "ContinuousBackupsStatus": "string",
      "PointInTimeRecoveryDescription": {
         "PointInTimeRecoveryStatus": "string",
         "EarliestRestorableDateTime": number,
         "RecoveryPeriodInDays": number,
```

```
          "LatestRestorableDateTime": number
       }
    }
}
```

Python

```
import boto3

dynamodb = boto3.client('dynamodb')

response = dynamodb.update_continuous_backups(
    TableName=<table_name>,
    PointInTimeRecoverySpecification={
        'PointInTimeRecoveryEnabled': True,
        'RecoveryPeriodInDays': 35
    }
)
```

Recovery Period

You can set the recovery period for continuous backups to be any number between 1 and 35 days. This `RecoveryPeriodInDays` determines the time period for which your continuous backups are maintained. For example, if you set this value to be 30 days, you'll only have the ability to restore your table to any point in time from the past 30 days.

> **Note**
>
> DynamoDB charges for PITR based on the size of each DynamoDB table, including table data and local secondary indexes. The configured maximum recovery period doesn't impact the price you're charged for turning on PITR. For details on pricing, see DynamoDB pricing.

Edit PITR

You can edit the PITR setting on your table and change the recovery period. If you change the recovery period and increase it to a value higher than previously set, your `EarliestRestorePoint` will not change immediately. Since the recovery period is a rolling window, DynamoDB will continue to take automatic backups until the new increased period reached. If you change the recovery period and decrease it to a value lower than previously set,

your `EarliestRestorePoint` will immediately decrease to match your recovery period, and any continuous backups that fall outside of the new set value will not be recoverable.

Delete a table with PITR enabled

When you delete a table that has point-in-time recovery enabled, DynamoDB automatically creates a backup snapshot called a *system backup* and retains it for 35 days (at no additional cost). You can use the system backup to restore the deleted table to the state it was in before deletion. All system backups follow a standard naming convention of *`table-name`*`$DeletedTableBackup`.

> **Note**
>
> Once a table with point-in-time recovery enabled is deleted, you can use system backup to restore that table to a single point in time. The system backup will be created upon table deletion, and is a snapshot of the table right before the table is deleted.

Using on-demand DynamoDB backup and restore

Amazon DynamoDB supports stand-alone on-demand backup and restore features. Those features are available to you independent of whether you use AWS Backup.

You can use the DynamoDB on-demand backup capability to create full backups of your tables for long-term retention and archival for regulatory compliance needs. You can back up and restore your table data anytime with a single click on the AWS Management Console or with a single API call. Backup and restore actions run with zero impact on table performance or availability.

You can create table backups using the console, the AWS Command Line Interface (AWS CLI), or the DynamoDB API. For more information, see Backing up a DynamoDB table.

For information about restoring a table from a backup, see Restoring a DynamoDB table from a backup.

Backing up and restoring DynamoDB tables with DynamoDB: How it works

You can use the DynamoDB on-demand backup feature to create full backups of your Amazon DynamoDB tables. This feature is available independently from AWS backup. This section provides an overview of what happens during the DynamoDB backup and restore process.

Backups

When you create an on-demand backup with DynamoDB, a time marker of the request is cataloged. The backup is created asynchronously by applying all changes until the time of the request to the last full table snapshot. DynamoDB backup requests are processed instantaneously and become available for restore within minutes.

> **Note**
>
> Each time you create an on-demand backup, the entire table data is backed up. There is no limit to the number of on-demand backups that can be taken.

All backups in DynamoDB work without consuming any provisioned throughput on the table.

DynamoDB backups do not guarantee causal consistency across items; however, the skew between updates in a backup is usually much less than a second.

While a backup is in progress, you can't do the following:

- Pause or cancel the backup operation.
- Delete the source table of the backup.
- Disable backups on a table if a backup for that table is in progress.

If you don't want to create scheduling scripts and cleanup jobs, you can use AWS Backup to create backup plans with schedules and retention policies for your DynamoDB tables. AWS Backup runs the backups and deletes them when they expire. For more information, see the AWS Backup Developer Guide.

In addition to AWS Backup, you can schedule periodic or future backups by using AWS Lambda functions. For more information, see the blog post A serverless solution to schedule your Amazon DynamoDB On-Demand backup.

If you're using the console, any backups created using AWS Backup are listed on the **Backups** tab with the **Backup type** set to `AWS`.

> **Note**
>
> You can't delete backups marked with a **Backup type** of AWS using the DynamoDB console. To manage these backups, use the AWS Backup console.

To learn how to perform a backup, see Backing up a DynamoDB table.

Restores

You restore a table without consuming any provisioned throughput on the table. You can do a full table restore from your DynamoDB backup, or you can configure the destination table settings. When you do a restore, you can change the following table settings:

- Global secondary indexes (GSIs)
- Local secondary indexes (LSIs)
- Billing mode
- Provisioned read and write capacity
- Encryption settings

> **Important**
>
> When you do a full table restore, the destination table is set with the same provisioned read capacity units and write capacity units as the source table, as recorded at the time the backup was requested. The restore process also restores the local secondary indexes and the global secondary indexes.

You can also restore your DynamoDB table data across AWS Regions such that the restored table is created in a different Region from where the backup resides. You can do cross-Region restores between AWS commercial Regions, AWS China Regions, and AWS GovCloud (US) Regions. You pay only for the data that you transfer out of the source Region and for restoring to a new table in the destination Region.

Restores can be faster and more cost-efficient if you choose to exclude some or all secondary indexes from being created on the new restored table.

You must manually set up the following on the restored table:

- Auto scaling policies
- AWS Identity and Access Management (IAM) policies
- Amazon CloudWatch metrics and alarms
- Tags
- Stream settings
- Time to Live (TTL) settings
- Deletion protection settings
- Point in time recovery (PITR) settings

You can only restore the entire table data to a new table from a backup. You can write to the restored table only after it becomes active.

> **Note**
>
> You can't overwrite an existing table during a restore operation.

Restore times are directly related to the configuration of your tables (such as the size of your tables and the number of underlying partitions) and other related variables. A best practice when planning for disaster recovery is to regularly document average restore completion times and establish how these times affect your overall Recovery Time Objective.

To learn how to perform a restore, see Restoring a DynamoDB table from a backup.

You can use IAM policies for access control. For more information, see Using IAM with DynamoDB backup and restore.

All backup and restore console and API actions are captured and recorded in AWS CloudTrail for logging, continuous monitoring, and auditing.

Backing up a DynamoDB table

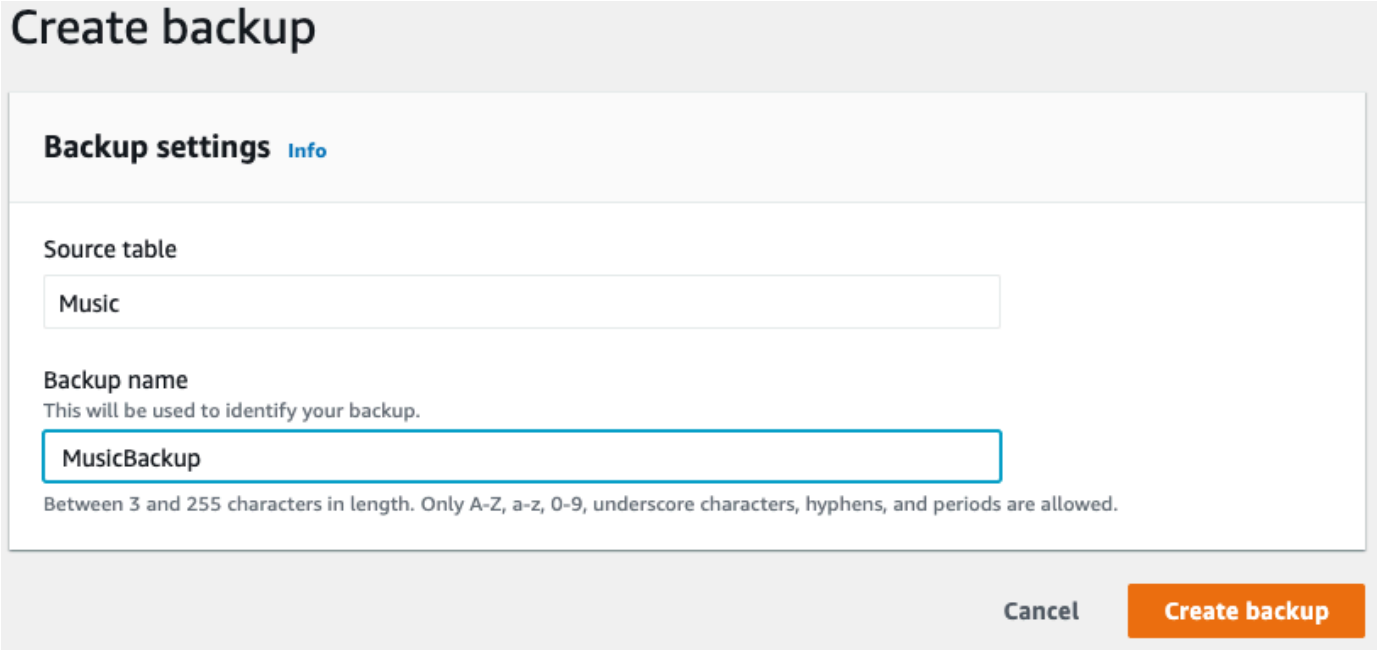
This section describes how to use the Amazon DynamoDB console or the AWS Command Line Interface to back up a table.

Creating a table backup (console)

Follow these steps to create a backup named `MusicBackup` for an existing `Music` table using the AWS Management Console.

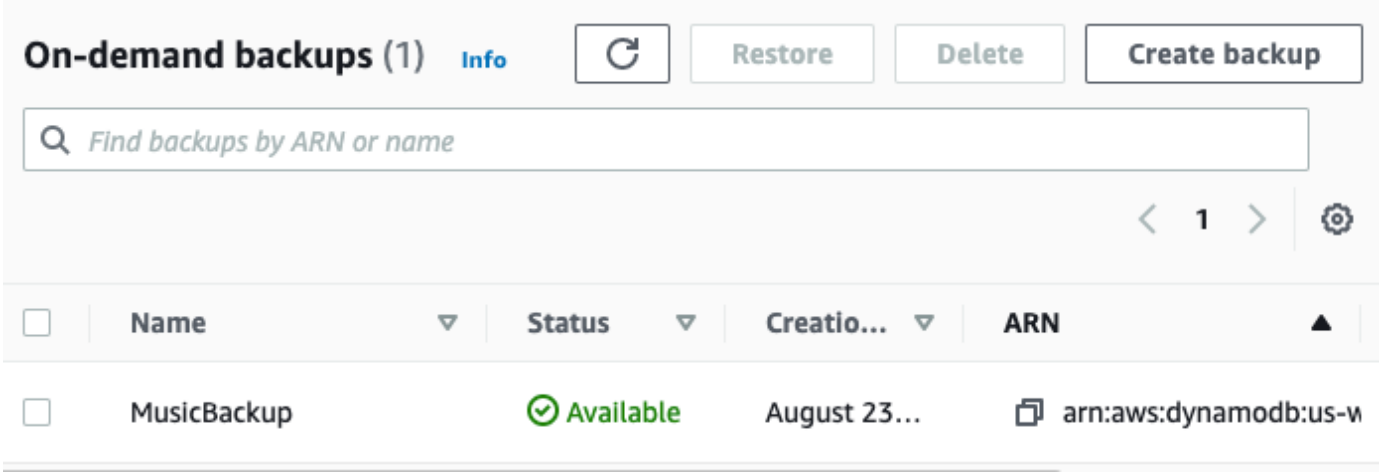
To create a table backup

1. Sign in to the AWS Management Console and open the DynamoDB console at https://console.aws.amazon.com/dynamodb/.
2. You can create a backup by doing one of the following:
   - On the **Backups** tab of the `Music` table, choose **Create backup**.
   - In the navigation pane on the left side of the console, choose **Backups**. Then choose **Create backup**.
3. Make sure that `Music` is the table name, and enter **`MusicBackup`** for the backup name. Then, choose **Create backup** to create the backup.



> **Note**
>
> If you create backups using the **Backups** section in the navigation pane, the table isn't preselected for you. You have to manually choose the source table name for the backup.

While the backup is being created, the backup status is set to **Creating**. After the backup is complete, the backup status changes to **Available**.



Creating a table backup (AWS CLI)

Follow these steps to create a backup for an existing table `Music` using the AWS CLI.

To create a table backup

- Create a backup with the name `MusicBackup` for the `Music` table.

```
aws dynamodb create-backup --table-name Music \
 --backup-name MusicBackup
```

While the backup is being created, the backup status is set to `CREATING`.

```
{
    "BackupDetails": {
        "BackupName": "MusicBackup",
        "BackupArn": "arn:aws:dynamodb:us-east-1:123456789012:table/Music/
backup/01489602797149-73d8d5bc",
        "BackupStatus": "CREATING",
        "BackupCreationDateTime": 1489602797.149
    }
}
```

After the backup is complete, its `BackupStatus` should change to `AVAILABLE`. To confirm this, use the `describe-backup` command. You can get the input value of `backup-arn` from the output of the previous step or by using the `list-backups` command.

```
aws dynamodb describe-backup \
--backup-arn arn:aws:dynamodb:us-east-1:123456789012:table/Music/backup/01489173575360-b308cd7d
```

To keep track of your backups, you can use the `list-backups` command. It lists all your backups that are in `CREATING` or `AVAILABLE` status.

```
aws dynamodb list-backups
```

The `list-backups` command and the `describe-backup` command are useful to check information about the source table of the backup.

Restoring a DynamoDB table from a backup

This section describes how to restore a table from a backup using the Amazon DynamoDB console or the AWS Command Line Interface (AWS CLI).

> **Note**
>
> If you want to use the AWS CLI, you must configure it first. For more information, see Accessing DynamoDB.

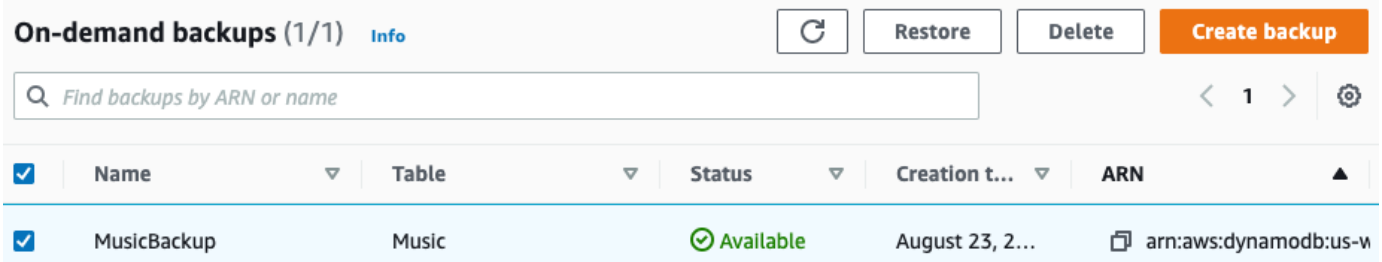
Restoring a table from a backup (console)

The following procedure shows how to restore the `Music` table by using the `MusicBackup` file that is created in the Backing up a DynamoDB table tutorial.

> **Note**
>
> This procedure assumes that the `Music` table no longer exists before restoring it using the `MusicBackup` file.

To restore a table from a backup

1. Sign in to the AWS Management Console and open the DynamoDB console at https://console.aws.amazon.com/dynamodb/.
2. In the navigation pane on the left side of the console, choose **Backups**.
3. In the list of backups, choose `MusicBackup`.



4. Choose **Restore**.
5. Enter **`Music`** as the new table name. Confirm the backup name and other backup details. Then choose **Restore table** to start the restore process.

> **Note**
>
> You can restore the table to the same AWS Region or to a different Region from where the backup resides. You can also exclude secondary indexes from being created on the new restored table. In addition, you can specify a different encryption mode.
> Tables restored from backups are always created using the DynamoDB Standard table class.

Restore table from backup Info

Restoring a table from a backup will restore it as a new table.

Restore settings

Name of restored table

This name will identify your restored table.

Music

Between 3 and 255 characters in length. Only A–Z, a–z, 0–9, underscore characters, hyphens, and periods allowed.

Secondary indexes

- (●) Restore the entire table
  Your restored table will include all local and global secondary indexes.
- ( ) Restore the table without secondary indexes
  Your restored table will exclude all local and global secondary indexes. Restoring this way can be faster and more cost efficient.

Destination AWS Region

- (●) Same Region (Oregon)
  Restore the table to the same Region as the original table.
- ( ) Cross-Region
  Restore the table to a different Region for greater redundancy but with higher data transfer costs.

▼ Encryption at rest - *optional*

All user data stored in Amazon DynamoDB is fully encrypted at rest. By default, Amazon DynamoDB manages the encryption key, and you are not charged any fee for using it.

Encryption key management Info

- (●) Owned by Amazon DynamoDB
  The key is owned and managed by DynamoDB. You are not charged an additional fee for using this customer master key (CMK).
- ( ) AWS managed CMK
  The key is stored in your account and is managed by AWS Key Management Service (AWS KMS). AWS KMS charges apply.
- ( ) Stored in your account, and owned and managed by you
  Choose a key that is owned and managed by you, and stored in AWS KMS.

ⓘ The time it takes to restore a table from a backup can vary and is based on multiple variables. After your table is restored from the backup, you might need to reapply configuration settings. Learn more

Cancel Restore

The table that is being restored is shown with the status **Creating**. After the restore process is finished, the status of the `Music` table changes to **Active**.

Restoring a table from a backup (AWS CLI)

Follow these steps to use the AWS CLI to restore the `Music` table using the `MusicBackup` that is created in the Backing up a DynamoDB table tutorial.

To restore a table from a backup

1. Confirm the backup that you want to restore by using the `list-backups` command. This example uses `MusicBackup`.

   ```
   aws dynamodb list-backups
   ```

   To get additional details for the backup, use the `describe-backup` command. You can get the input `backup-arn` from the previous step.

   ```
   aws dynamodb describe-backup \
   --backup-arn arn:aws:dynamodb:us-east-1:123456789012:table/Music/
   backup/01489173575360-b308cd7d
   ```

2. Restore the table from the backup. In this case, the `MusicBackup` restores the `Music` table to the same AWS Region.

   ```
   aws dynamodb restore-table-from-backup \
   --target-table-name Music \
   --backup-arn arn:aws:dynamodb:us-east-1:123456789012:table/Music/
   backup/01489173575360-b308cd7d
   ```

3. Restore the table from the backup with custom table settings. In this case, the `MusicBackup` restores the `Music` table and specifies an encryption mode for the restored table.

   > **Note**
   >
   > The `sse-specification-override` parameter takes the same values as the `sse-specification-override` parameter used in the `CreateTable` command. To learn more, see Managing encrypted tables in DynamoDB.

```
aws dynamodb restore-table-from-backup \
--target-table-name Music \
--backup-arn arn:aws:dynamodb:us-east-1:123456789012:table/Music/
backup/01581080476474-e177ebe2 \
--sse-specification-override Enabled=true,SSEType=KMS,KMSMasterKeyId=abcd1234-
abcd-1234-a123-ab1234a1b234
```

You can restore the table to a different AWS Region from where the backup resides.

Note

- The `sse-specification-override` parameter is mandatory for cross-Region restores but optional for restores in the same Region as the source table.
- When performing a cross-Region restore from the command line, you must set the default AWS Region to the desired destination Region. To learn more, see Command line options in the *AWS Command Line Interface User Guide*.

```
aws dynamodb restore-table-from-backup \
--target-table-name Music \
--backup-arn arn:aws:dynamodb:us-east-1:123456789012:table/Music/
backup/01581080476474-e177ebe2 \
--sse-specification-override Enabled=true,SSEType=KMS
```

You can override the billing mode and the provisioned throughput for the restored table.

```
aws dynamodb restore-table-from-backup \
--target-table-name Music \
--backup-arn arn:aws:dynamodb:us-east-1:123456789012:table/Music/
backup/01489173575360-b308cd7d \
--billing-mode-override PAY_PER_REQUEST
```

You can exclude some or all secondary indexes from being created on the restored table.

> **Note**
>
> Restores can be faster and more cost-efficient if you exclude some or all secondary indexes from being created on the restored table.

```
aws dynamodb restore-table-from-backup \
--target-table-name Music \
--backup-arn arn:aws:dynamodb:us-east-1:123456789012:table/Music/
backup/01581081403719-db9c1f91 \
--global-secondary-index-override '[]' \
--sse-specification-override Enabled=true,SSEType=KMS
```

> **Note**
>
> The secondary indexes provided should match existing indexes. You cannot create new indexes at the time of restore.

You can use a combination of different overrides. For example, you can use a single global secondary index and change provisioned throughput at the same time, as follows.

```
aws dynamodb restore-table-from-backup \
--target-table-name Music \
--backup-arn arn:aws:dynamodb:eu-west-1:123456789012:table/Music/
backup/01581082594992-303b6239 \
--billing-mode-override PROVISIONED \
--provisioned-throughput-override ReadCapacityUnits=100,WriteCapacityUnits=100 \
--global-secondary-index-override IndexName=singers-
index,KeySchema=["{AttributeName=SingerName,KeyType=HASH}"],Projection="{ProjectionType=KEYS_
 \
--sse-specification-override Enabled=true,SSEType=KMS
```

To verify the restore, use the `describe-table` command to describe the `Music` table.

```
aws dynamodb describe-table --table-name Music
```

The table that is being restored from the backup is shown with the status **Creating**. After the restore process is finished, the status of the `Music` table changes to **Active**.

> **Important**
>
> While a restore is in progress, don't modify or delete your IAM role policy; otherwise, unexpected behavior can result. For example, suppose that you removed write permissions for a table while that table is being restored. In this case, the underlying `RestoreTableFromBackup` operation would not be able to write any of the restored data to the table.
> After the restore operation is complete, you can modify or delete your IAM role policy.
> IAM policies involving source IP restrictions for accessing the target restore table should have the `aws:ViaAWSService` key set to `false` to ensure that the restrictions apply only to requests made directly by a principal. Otherwise, the restore will be canceled.
> If your backup is encrypted with an AWS managed key or a customer managed key, don't disable or delete the key while a restore is in progress, or the restore will fail.
> After the restore operation is complete, you can change the encryption key for the restored table and disable or delete the old key.

Deleting a DynamoDB table backup

This section describes how to use the AWS Management Console or the AWS Command Line Interface (AWS CLI) to delete an Amazon DynamoDB table backup.

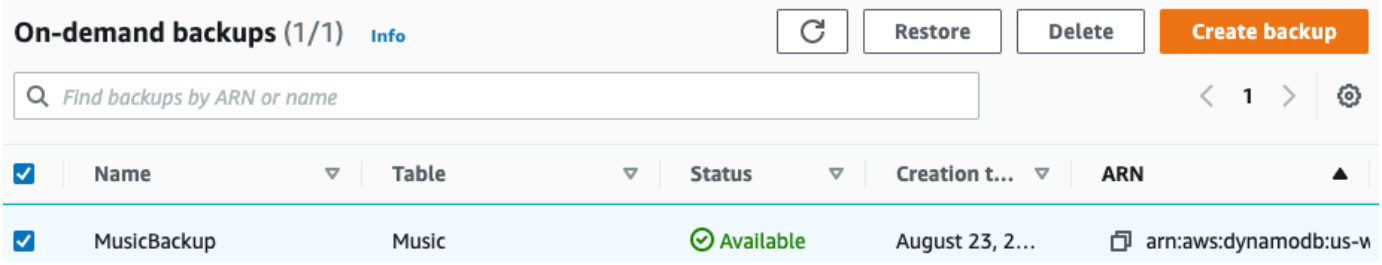
> **Note**
>
> If you want to use the AWS CLI, you have to configure it first. For more information, see Using the AWS CLI.

Deleting a table backup (console)

The following procedure shows how to use the console to delete the `MusicBackup` that is created in the Backing up a DynamoDB table tutorial.

To delete a backup

1. Sign in to the AWS Management Console and open the DynamoDB console at https://console.aws.amazon.com/dynamodb/.
2. In the navigation pane on the left side of the console, choose **Backups**.
3. In the list of backups, choose `MusicBackup`.



4. Choose **Delete**. Confirm that you want to delete the backup by typing **`delete`** and clicking **Delete**.

Deleting a table backup (AWS CLI)

The following example deletes a backup for an existing table `Music` table using the AWS CLI.

```
aws dynamodb delete-backup \
--backup-arn arn:aws:dynamodb:us-east-1:123456789012:table/Music/
backup/01489602797149-73d8d5bc
```

Using IAM with DynamoDB backup and restore

You can use AWS Identity and Access Management (IAM) to restrict Amazon DynamoDB backup and restore actions for some resources. The `CreateBackup` and `RestoreTableFromBackup` APIs operate on a per-table basis.

For more information about using IAM policies in DynamoDB, see Identity-based policies for DynamoDB.

The following are examples of IAM policies that you can use to configure specific backup and restore functionality in DynamoDB.

Example 1: Allow the CreateBackup and RestoreTableFromBackup actions

The following IAM policy grants permissions to allow the `CreateBackup` and `RestoreTableFromBackup` DynamoDB actions on all tables:

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "dynamodb:CreateBackup",
                "dynamodb:RestoreTableFromBackup",
                "dynamodb:PutItem",
                "dynamodb:UpdateItem",
                "dynamodb:DeleteItem",
                "dynamodb:GetItem",
                "dynamodb:Query",
                "dynamodb:Scan",
                "dynamodb:BatchWriteItem"
            ],
            "Resource": "*"
        }
    ]
}
```

Important

DynamoDB RestoreTableFromBackup permissions are necessary on the source backup, and DynamoDB read and write permissions on the target table are necessary for restore functionality.
DynamoDB RestoreTableToPointInTime permissions are necessary on the source table, and DynamoDB read and write permissions on the target table are necessary for restore functionality.

Example 2: Allow CreateBackup and deny RestoreTableFromBackup

The following IAM policy grants permissions for the `CreateBackup` action and denies the `RestoreTableFromBackup` action:

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": ["dynamodb:CreateBackup"],
            "Resource": "*"
        },
        {
            "Effect": "Deny",
            "Action": ["dynamodb:RestoreTableFromBackup"],
            "Resource": "*"
        }

    ]
}
```

Example 3: Allow ListBackups and deny CreateBackup and RestoreTableFromBackup

The following IAM policy grants permissions for the `ListBackups` action and denies the `CreateBackup` and `RestoreTableFromBackup` actions:

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": ["dynamodb:ListBackups"],
            "Resource": "*"
        },
        {
            "Effect": "Deny",
            "Action": [
                "dynamodb:CreateBackup",
                "dynamodb:RestoreTableFromBackup"
```

```
            ],
            "Resource": "*"
        }

    ]
}
```

Example 4: Allow ListBackups and deny DeleteBackup

The following IAM policy grants permissions for the `ListBackups` action and denies the `DeleteBackup` action:

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": ["dynamodb:ListBackups"],
            "Resource": "*"
        },
        {
            "Effect": "Deny",
            "Action": ["dynamodb:DeleteBackup"],
            "Resource": "*"
        }

    ]
}
```

Example 5: Allow RestoreTableFromBackup and DescribeBackup for all resources and deny DeleteBackup for a specific backup

The following IAM policy grants permissions for the `RestoreTableFromBackup` and `DescribeBackup` actions and denies the `DeleteBackup` action for a specific backup resource:

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "dynamodb:DescribeBackup",
                "dynamodb:RestoreTableFromBackup"
            ],
            "Resource": "arn:aws:dynamodb:us-east-1:123456789012:table/Music/
backup/01489173575360-b308cd7d"
        },
        {
            "Effect": "Allow",
            "Action": [
                "dynamodb:PutItem",
                "dynamodb:UpdateItem",
                "dynamodb:DeleteItem",
                "dynamodb:GetItem",
                "dynamodb:Query",
                "dynamodb:Scan",
                "dynamodb:BatchWriteItem"
            ],
            "Resource": "*"
        },
        {
            "Effect": "Deny",
            "Action": [
                "dynamodb:DeleteBackup"
            ],
            "Resource": "arn:aws:dynamodb:us-east-1:123456789012:table/Music/
backup/01489173575360-b308cd7d"
        }
    ]
}
```

Important

DynamoDB RestoreTableFromBackup permissions are necessary on the source backup, and DynamoDB read and write permissions on the target table are necessary for restore functionality.
DynamoDB RestoreTableToPointInTime permissions are necessary on the source table, and DynamoDB read and write permissions on the target table are necessary for restore functionality.

Example 6: Allow CreateBackup for a specific table

The following IAM policy grants permissions for the `CreateBackup` action on the `Movies` table only:

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": ["dynamodb:CreateBackup"],
            "Resource": [
                "arn:aws:dynamodb:us-east-1:123456789012:table/Movies"
            ]
        }
    ]
}
```

Example 7: Allow ListBackups

The following IAM policy grants permissions for the `ListBackups` action:

JSON

```
{
    "Version":"2012-10-17",
```

```
    "Statement": [
        {
            "Effect": "Allow",
            "Action": ["dynamodb:ListBackups"],
            "Resource": "*"
        }
    ]
}
```

> **Important**
>
> You can't grant permissions for the `ListBackups` action on a specific table.

Example 8: Allow access to AWS Backup features

You will need API permissions for the `StartAwsBackupJob` action for a successful backup with advanced features, and the `dynamodb:RestoreTableFromAwsBackup` action to successfully restore that backup.

The following IAM policy grants AWS Backup the permissions to trigger backups with advanced features and restores. Also note that if the tables are encrypted the policy will need access to the AWS KMS key.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "DescribeQueryScanBooksTable",
            "Effect": "Allow",
            "Action": [
                "dynamodb:StartAwsBackupJob",
                "dynamodb:DescribeTable",
                "dynamodb:Query",
                "dynamodb:Scan"
            ],
            "Resource": "arn:aws:dynamodb:us-west-2:111122223333:table/Books"
        },
```

```
        {
            "Sid": "AllowRestoreFromAwsBackup",
            "Effect": "Allow",
            "Action": [
                "dynamodb:RestoreTableFromAwsBackup"
            ],
            "Resource": "*"
        }
    ]
}
```

Example 9: Deny RestoreTableToPointInTime for a Specific Source Table

The following IAM policy denies permissions for the `RestoreTableToPointInTime` action for a specific source table:

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Deny",
            "Action": [
                "dynamodb:RestoreTableToPointInTime"
            ],
            "Resource": "arn:aws:dynamodb:us-east-1:123456789012:table/Music"
        }
    ]
}
```

Example 10: Deny RestoreTableFromBackup for all Backups for a Specific Source Table

The following IAM policy denies permissions for the `RestoreTableToPointInTime` action for all backups for a specific source table:

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Deny",
            "Action": [
                "dynamodb:RestoreTableFromBackup"
            ],
            "Resource": "arn:aws:dynamodb:us-east-1:123456789012:table/Music/
backup/*"
        }
    ]
}
```

Understanding Amazon DynamoDB billing for backups

This guide provides details about how DynamoDB billing works for backups. We'll break down the various components that contribute to the overall cost, providing clear explanations and practical examples.

DynamoDB offers on-demand backups and point-in-time recovery (PITR) backups to help protect your DynamoDB data from disaster events and offers data archiving for long-term retention.

How it works

DynamoDB on-demand backups are billed monthly. If you take a backup on any particular day of the month, you'll see a single charge for that backup calculated for the remaining days of the month (example: creating a backup on the 27th, you will only be charged for the few days remaining in that month, applied as a single charge on the 27th).

If you retain your previously taken backups for subsequent months, you'll always see a full month's charge for that backup applied on the 1st. If the backup is removed before the month's end, the charges will be adjusted based on actual usage.

As an example, if you created a backup on July 27th, and it is maintained through the month of August, you will see the following charges for that backup:

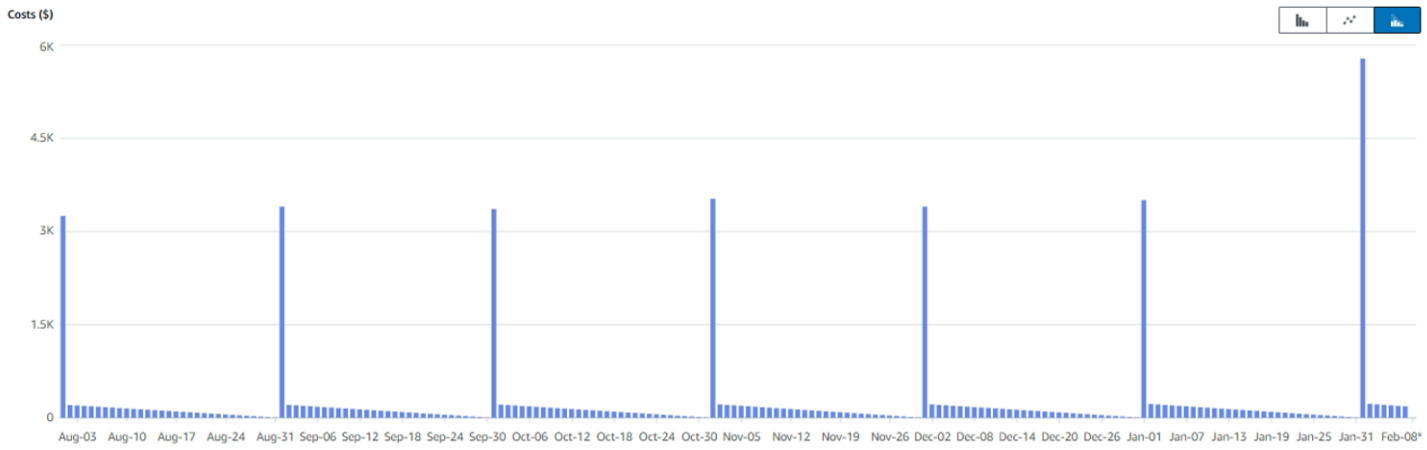
- A charge on July 27th for the remaining days of July
- A charge on August 1st for the entire month of August
- A charge on the 1st of every subsequent month that backup exists

When backups are maintained for DynamoDB tables, you may observe that the expense for the `DynamoDB (Region)-TimedBackupStorage-ByteHrs` usage metric seems abnormally high on the 1st of the month. In addition, if you check this metric at the start of a new month and compare it against previous billing cycles, you may observe what appears to be a large spike in usage. This is by design. On the 1st of every month, any existing DynamoDB backups will have usage charges for the entire month applied. Any DynamoDB backups that are removed during the month will have their usage expense prorated to reflect actual usage. As a result, you may see the charge (applied on the 1st) decrease throughout the month. This is because retention policies apply expirations or manual deletions to carried over backups occur. This will be explored in a scenario below.

Similarly, you'll notice smaller spikes throughout the month as new backups are created, with their charges applied on the day of creation for the remainder of the month.

DynamoDB backup billing example

Here is an example of what you may see in Cost Explorer at the start of the month:



Notice how February 1st appears to have a much larger spike relative to previous months. Let's break down why this occurs.

From the DynamoDB Pricing Page:

"The total backup storage size billed each month is the sum of all backups of DynamoDB tables. DynamoDB monitors the size of on-demand backups continuously throughout the month to determine your backup charges."

This explains why the bill consistently shows a large spike in usage on the 1st of every month. Any existing backups coming into a new month have a full month's charges applied to the 1st. Put another way, if you enter the month with 300 DynamoDB backups, you will see a full month's usage charges applied on the 1st day of the month for all 300 backups.

New backups taken throughout the month will incur usage charges from the day of creation until the end of the month.

What if the backup is deleted mid-month?

Here are a few scenarios to consider:

1. If a backup from the previous month is deleted on the 15th of the current month, the usage charges for that backup,still applied on the 1st, will be adjusted to the actual usage instead of the assumed full month of usage previously applied. The example below explains this more in detail.
2. When you create a backup during the month, usage charges for the remainder of the month are applied to the day it was created. However, if you delete this backup before the month ends, your usage charges will be adjusted to include only the days the backup was active, still applied on its creation date.

Why does the current month's usage appear to be so much higher on the 1st than previous months, and what happens if I remove the backups?

To answer this important 2-part question, let's set up an example scenario using the following information:

- **Length of Month**: 30 Days
- **DynamoDB Backup Frequency**: 10/day, 300/mo
- **DynamoDB Backup Retention Policy**: 30 Days
- **DynamoDB Per-Backup Cost**: $2/day, $60/mo
- **Previous 1st of Month Total** (`TimedBackupStorage-ByteHrs`, checked on the 1st of the Current Month): $9,300
- **Previous Month Total** (`TimedBackupStorage-ByteHrs`): $18,600

- **Current 1st of Month Total** (`TimedBackupStorage-ByteHrs`, Checked on the 1st): $18,000
- **Changes in DynamoDB Usage Month-to-Month**: None

Using the information above, we can see that 300 backups were created in the previous month with a policy to maintain them for 30 days. On the 1st of a new month, all of these backups still remain as they've not yet hit the end of their recovery period. However, with each passing day, the oldest sets of backups will begin to drop off, as shown here:

DynamoDB backup dropoff table

New month	Day 1	Day 2	Day 3	Day 4	Day 5
Total previous month backups carried over	300	290	280	270	260

- On the 1st, we can see 300 backups @ $60/mo per backup, totaling $18,000 of `TimedBackupStorage-ByteHrs` applied. This is in contrast to the previous month, where the entire month's total was $18,600.
- On the 2nd, 10 of those backups will have expired and drop off. When this occurs, the applied charge for those backups will be adjusted to **Actual Usage** instead of **Assumed Usage**. This results in those 10 backups, previously with an applied charge on the 1st of $600 (10 Backups x 30 Days) being adjusted down to $20 (10 Backups x 1 Day).
- The following day, the next block of 10 will expire and drop, shifting their usage from 30 days down to 2 days, reducing their charge to $40 (10 Backups x 2 Days).

With every passing day, we'll see that larger-than-previous-month spike begin to shrink. If we expand this to cover the entire month, we'll observe the following:

DynamoDB Backup Charges (1st of Month) Progression

300 backups in blocks of 10	1st	10th	20th	30th
Block 1	$600	$20	$20	$20

300 backups in blocks of 10	1st	10th	20th	30th
Block 2	$600	$40	$40	$40
Block 3	$600	$60	$60	$60
Block 4	$600	$80	$80	$80
Block 5	$600	$100	$100	$100
Block 6	$600	$120	$120	$120
Block 7	$600	$140	$140	$140
Block 8	$600	$160	$160	$160
Block 9	$600	$180	$180	$180
Block 10	$600	$600	$200	$200
Block 11	$600	$600	$220	$220
Block 12	$600	$600	$240	$240
Block 13	$600	$600	$260	$260
Block 14	$600	$600	$280	$280
Block 15	$600	$600	$300	$300
Block 16	$600	$600	$320	$320
Block 17	$600	$600	$340	$340
Block 18	$600	$600	$360	$360
Block 19	$600	$600	$380	$380
Block 20	$600	$600	$600	$400
Block 21	$600	$600	$600	$420

300 backups in blocks of 10	1st	10th	20th	30th
Block 22	$600	$600	$600	$440
Block 23	$600	$600	$600	$460
Block 24	$600	$600	$600	$480
Block 25	$600	$600	$600	$500
Block 26	$600	$600	$600	$520
Block 27	$600	$600	$600	$540
Block 28	$600	$600	$600	$560
Block 29	$600	$600	$600	$580
Block 30	$600	$600	$600	$600
1st of month total ($)	$18,000	$13,500	$10,400	$9,300

As a new block drops off each day, it has its usage adjusted to how many days it existed, versus the full month amount. As a result, by month's end the charges observed on the 1st will have dropped from the initial $18,000 down to the expected $9,300. This number, combined with the newly created backups through the month (which will have a billing table similar to the above, but reversed), will result in a monthly expense in line with last month's $18,600.

Restore a table in DynamoDB

You can restore a DynamoDB table from your PITR backup or your on-demand backups using the AWS Management Console, the AWS Command Line Interface (AWS CLI), or the DynamoDB API. The recovery process restores to a new DynamoDB table.

Restoring a table using point-in-time recovery

You can restore your table to any point in time up till the `EarliestRestoreableDateTime`.

Important

If you disable point-in-time recovery and later enable it on a table, you reset the start time for which you can recover that table. As a result, you can only immediately restore that table using the `LatestRestorableDateTime`.

When you restore using point-in-time recovery, DynamoDB restores your table data to the state based on the selected date and time (day:hour:minute:second) to a new table. You restore a table without consuming any provisioned throughput on the table. You can do a full table restore using point-in-time recovery, or you can configure the destination table settings. You can change the following table settings on the restored table:

- Global secondary indexes (GSIs)
- Local secondary indexes (LSIs)
- Billing mode
- Provisioned read and write capacity
- Encryption settings

Important

When you do a full table restore, the destination table is set with the same provisioned read capacity units and write capacity units that the source table had when the backup was requested. For example, suppose that a table's provisioned throughput was recently lowered to 50 read capacity units and 50 write capacity units. You then restore the table's state to three weeks ago, at which time its provisioned throughput was set to 100 read capacity units and 100 write capacity units. In this case, DynamoDB restores your table data to that point in time with the provisioned throughput from that time (100 read capacity units and 100 write capacity units).

You can also restore your DynamoDB table data across AWS Regions such that the restored table is created in a different Region from where the source table resides. You can do cross-Region restores between AWS commercial Regions, AWS China Regions, and AWS GovCloud (US). You pay only for the data you transfer out of the source Region and for restoring to a new table in the destination Region.

> **Note**
>
> Cross-Region restore isn't supported if the source or destination Region is Asia Pacific (Hong Kong) or Middle East (Bahrain).

Restores can be faster and more cost-efficient if you exclude some or all indexes from being created on the restored table. You must manually set the following on the restored table:

- Auto scaling policies
- AWS Identity and Access Management policies
- Amazon CloudWatch Events metrics and alarms
- Tags
- Stream settings
- Time to Live (TTL) settings
- Point-in-time recovery settings

The time it takes you to restore a table varies based on multiple factors and isn't always correlated with the size of the table.

Restoring a DynamoDB table to a point in time

Amazon DynamoDB point-in-time recovery (PITR) provides continuous backups of your DynamoDB table data. You can restore a table to a point in time using the DynamoDB console or the AWS Command Line Interface (AWS CLI). The point-in-time recovery process restores to a new table.

If you want to use the AWS CLI, you must configure it first. For more information, see Accessing DynamoDB.

Topics

- Restoring a DynamoDB table to a point in time (console)
- Restoring a table to a point in time (AWS CLI)

Restoring a DynamoDB table to a point in time (console)

The following example demonstrates how to use the DynamoDB console to restore an existing table named `Music` to a point in time.

> **Note**
>
> This procedure assumes that you have enabled point-in-time recovery. To enable it for the `Music` table, on the **Backups** tab, in the **Point-in-time recovery (PITR)** section, choose **Edit** and then check the box next to **Enable point-in-time-recovery**.

To restore a table to a point in time

1. Sign in to the AWS Management Console and open the DynamoDB console at https://console.aws.amazon.com/dynamodb/.
2. In the navigation pane on the left side of the console, choose **Tables**.
3. In the list of tables, choose the `Music` table.
4. On the **Backups** tab of the `Music` table, in the **Point-in-time recovery (PITR)** section, choose **Restore**.
5. For the new table name, enter **`MusicMinutesAgo`**.

   > **Note**
   >
   > You can restore the table to the same AWS Region or to a different Region from where the source table resides. You can also exclude secondary indexes from being created on the restored table. In addition, you can specify a different encryption mode.

6. To confirm the restorable time, set the restore date and time to **Earliest**. Then choose **Restore** to start the restore process.

   The table that is being restored is shown with the status **Restoring**. After the restore process is finished, the status of the `MusicMinutesAgo` table changes to **Active**.

Restoring a table to a point in time (AWS CLI)

The following procedure shows how to use the AWS CLI to restore an existing table named `Music` to a point in time.

Note

This procedure assumes that you have enabled point-in-time recovery. To enable it for the `Music` table, run the following command.

```
aws dynamodb update-continuous-backups \
    --table-name Music \
    --point-in-time-recovery-specification PointInTimeRecoveryEnabled=True
```

To restore a table to a point in time

1. Confirm that point-in-time recovery is enabled for the `Music` table by using the `describe-continuous-backups` command.

   ```
   aws dynamodb describe-continuous-backups \
       --table-name Music
   ```

   Continuous backups (automatically enabled on table creation) and point-in-time recovery are enabled.

   ```
   {
       "ContinuousBackupsDescription": {
           "PointInTimeRecoveryDescription": {
               "PointInTimeRecoveryStatus": "ENABLED",
               "EarliestRestorableDateTime": 1519257118.0,
               "LatestRestorableDateTime": 1520018653.01
           },
           "ContinuousBackupsStatus": "ENABLED"
       }
   }
   ```

2. Restore the table to a point in time. In this case, the `Music` table is restored to the `LatestRestorableDateTime` (~5 minutes ago) to the same AWS Region.

   ```
   aws dynamodb restore-table-to-point-in-time \
       --source-table-name Music \
       --target-table-name MusicMinutesAgo \
       --use-latest-restorable-time
   ```

> **Note**
>
> You can also restore to a specific point in time. To do this, run the command using the `--restore-date-time` argument, and specify a timestamp. You can specify any point in time within the configured recovery period, which can be set to any value between 1 and 35 days. For example, the following command restores the table to the `EarliestRestorableDateTime`.
>
> ```
> aws dynamodb restore-table-to-point-in-time \
>     --source-table-name Music \
>     --target-table-name MusicEarliestRestorableDateTime \
>     --no-use-latest-restorable-time \
>     --restore-date-time 1519257118.0
> ```
>
> Specifying the `--no-use-latest-restorable-time` argument is optional when restoring to a specific point in time.

3. Restore the table to a point in time with custom table settings. In this case, the `Music` table is restored to the `LatestRestorableDateTime` (~5 minutes ago).

   You can specify a different encryption mode for the restored table, as follows.

   > **Note**
   >
   > The `sse-specification-override` parameter takes the same values as the `sse-specification-override` parameter used in the `CreateTable` command. To learn more, see Managing encrypted tables in DynamoDB.

   ```
   aws dynamodb restore-table-to-point-in-time \
       --source-table-name Music \
       --target-table-name MusicMinutesAgo \
       --use-latest-restorable-time \
       --sse-specification-override Enabled=true,SSEType=KMS,KMSMasterKeyId=abcd1234-
   abcd-1234-a123-ab1234a1b234
   ```

   You can restore the table to a different AWS Region from where the source table resides.

Note

- The `sse-specification-override` parameter is mandatory for cross-Region restores but optional for restores to the same Region as the source table.
- The `source-table-arn` parameter must be provided for cross-Region restores.
- When performing a cross-Region restore from the command line, you must set the default AWS Region to the desired destination Region. To learn more, see Command line options in the *AWS Command Line Interface User Guide*.

```
aws dynamodb restore-table-to-point-in-time \
    --source-table-arn arn:aws:dynamodb:us-east-1:123456789012:table/Music \
    --target-table-name MusicMinutesAgo \
    --use-latest-restorable-time \
    --sse-specification-override Enabled=true,SSEType=KMS,KMSMasterKeyId=abcd1234-
abcd-1234-a123-ab1234a1b234
```

You can override the billing mode and the provisioned throughput for the restored table.

```
aws dynamodb restore-table-to-point-in-time \
    --source-table-name Music \
    --target-table-name MusicMinutesAgo \
    --use-latest-restorable-time \
    --billing-mode-override PAY_PER_REQUEST
```

You can exclude some or all secondary indexes from being created on the restored table.

Note

Restores can be faster and more cost-efficient if you exclude some or all secondary indexes from being created on the new restored table.

```
aws dynamodb restore-table-to-point-in-time \
    --source-table-name Music \
    --target-table-name MusicMinutesAgo \
    --use-latest-restorable-time \
```

```
    --global-secondary-index-override '[]'
```

You can use a combination of different overrides. For example, you can use a single global secondary index and change provisioned throughput at the same time, as follows.

```
aws dynamodb restore-table-to-point-in-time \
    --source-table-name Music \
    --target-table-name MusicMinutesAgo \
    --billing-mode-override PROVISIONED \
    --provisioned-throughput-override ReadCapacityUnits=100,WriteCapacityUnits=100
 \
    --global-secondary-index-override IndexName=singers-
index,KeySchema=["{AttributeName=SingerName,KeyType=HASH}"],Projection="{ProjectionType=KEY
 \
    --sse-specification-override Enabled=true,SSEType=KMS \
    --use-latest-restorable-time
```

To verify the restore, use the `describe-table` command to describe the `MusicEarliestRestorableDateTime` table.

```
aws dynamodb describe-table --table-name MusicEarliestRestorableDateTime
```

The table that is being restored is shown with the status **Creating** and restore in progress as **true**. After the restore process is finished, the status of the `MusicEarliestRestorableDateTime` table changes to **Active**.

> **Important**
>
> While a restore is in progress, don't modify or delete the AWS Identity and Access Management (IAM) policies that grant the IAM entity (for example, user, group, or role) permission to perform the restore. Otherwise, unexpected behavior can result. For example, suppose that you remove write permissions for a table while that table is being restored. In this case, the underlying `RestoreTableToPointInTime` operation can't write any of the restored data to the table. IAM policies involving source IP restrictions for accessing the target restore table can similarly cause issues.
> You can modify or delete permissions only after the restore operation is completed.

Using AWS Backup with DynamoDB

Amazon DynamoDB can help you meet regulatory compliance and business continuity requirements through enhanced backup features in AWS Backup. AWS Backup is a fully managed data protection service that makes it easy to centralize and automate backups across AWS services, in the cloud, and on premises. Using this service, you can configure backup policies and monitor activity for your AWS resources in one place. To use AWS Backup, you must affirmatively opt-in. Opt-in choices apply to the specific account and AWS Region, so you might have to opt in to multiple Regions using the same account. For more information, see the AWS Backup Developer Guide.

Amazon DynamoDB is natively integrated with AWS Backup. You can use AWS Backup to schedule, copy, tag and life cycle your DynamoDB on-demand backups automatically. You can continue to view and restore these backups from the DynamoDB console. You can use the DynamoDB console, API, and AWS Command Line Interface (AWS CLI) to enable automatic backups for your DynamoDB tables.

> **Note**
>
> Any backups made through DynamoDB will remain unchanged. You will still be able to create backups through the current DynamoDB workflow.

Enhanced backup features available through AWS Backup include:

Scheduled backups - You can set up regularly scheduled backups of your DynamoDB tables using backup plans.

Cross-account and cross-Region copying - You can automatically copy your backups to another backup vault in a different AWS Region or account, which allows you to support your data protection requirements.

Cold storage tiering - You can configure your backups to implement life cycle rules to delete or transition backups to colder storage. This can help you optimize your backup costs.

Tags - You can automatically tag your backups for billing and cost allocation purposes.

Encryption – DynamoDB on-demand backups managed through AWS Backup are now stored in the AWS Backup vault. This allows you to encrypt and secure your backups by using an AWS KMS key that is independent from your DynamoDB table encryption key.

Audit backups – You can use AWS Backup Audit Manager to audit the compliance of your AWS Backup policies and to find backup activity and resources that are not yet compliant with the controls that you defined. You can also use it to automatically generate an audit trail of daily and on-demand reports for your backup governance purposes.

Secure backups using the WORM model – You can use AWS Backup Vault Lock to enable a write-once-read-many (WORM) setting for your backups. With AWS Backup Vault Lock, you can add an additional layer of defense that protects backups from inadvertent or malicious delete operations, changes to backup recovery periods, and updates to lifecycle settings. To learn more, see AWS Backup Vault Lock.

These enhanced backup features are available in all AWS Regions. To learn more about these features, see the AWS Backup Developer Guide.

Topics

- Backing up and restoring DynamoDB tables with AWS Backup: How it works
- Creating backups of DynamoDB tables with AWS Backup
- Copying a backup of a DynamoDB table with AWS Backup
- Restoring a backup of a DynamoDB table from AWS Backup
- Deleting a backup of a DynamoDB table with AWS Backup
- Usage note differences between on-demand backups managed by AWS Backup and DynamoDB

Backing up and restoring DynamoDB tables with AWS Backup: How it works

You can use the on-demand backup feature to create full backups of your Amazon DynamoDB tables. This section provides an overview of what happens during the backup and restore process.

Backups

When you create an on-demand backup with AWS Backup, a time marker of the request is cataloged. The backup is created asynchronously by applying all changes until the time of the request to the last full table snapshot.

Each time you create an on-demand backup, the entire table data is backed up. There is no limit to the number of on-demand backups that can be taken.

> **Note**
>
> Unlike DynamoDB Backups, backups made with AWS Backup are not instantaneous.

While a backup is in progress, you can't do the following:

- Pause or cancel the backup operation.
- Delete the source table of the backup.
- Disable backups on a table if a backup for that table is in progress.

AWS Backup provides automated backup schedules, retention management, and lifecycle management. This removes the need for custom scripts and manual processes. AWS Backup runs the backups and deletes them when they expire. For more information, see the AWS Backup Developer Guide.

If you're using the console, any backups created using AWS Backup are listed on the **Backups** tab with the **Backup type** set to `AWS_BACKUP`.

> **Note**
>
> You can't delete backups marked with a **Backup type** of `AWS_BACKUP` using the DynamoDB console. To manage these backups, use the AWS Backup console.

To learn how to perform a backup, see Backing up a DynamoDB table.

Restores

You restore a table without consuming any provisioned throughput on the table. You can do a full table restore from your DynamoDB backup, or you can configure the destination table settings. When you do a restore, you can change the following table settings:

- Encryption settings

Important

When you do a full table restore, the destination table is set with the same provisioned read capacity units and write capacity units that the source table had when the backup was requested. The restore process also restores the local secondary indexes and the global secondary indexes.

You can copy a backup of your DynamoDB table data to a different AWS Region and then restore it in that new Region. You can copy and then restore backups between AWS commercial Regions, AWS China Regions, and AWS GovCloud (US) Regions. You pay only for the data you copy from the source Region and the data you restore to a new table in the destination Region.

AWS Backup will restore the tables with all the original indexes.

You must manually set up the following on the restored table:

- Auto scaling policies
- AWS Identity and Access Management (IAM) policies
- Amazon CloudWatch metrics and alarms
- Tags
- Stream settings
- Time to Live (TTL) settings
- Deletion protection settings
- Point in time recovery (PITR) settings

You can only restore the entire table data to a new table from a backup. You can write to the restored table only after it becomes active.

Note

AWS Backup restores are nondestructive. You can't overwrite an existing table during a restore operation.

Restore times are directly related to the configuration of your tables (such as the size of your tables and the number of underlying partitions) and other related variables. A best practice when

planning for disaster recovery is to regularly document average restore completion times and establish how these times affect your overall Recovery Time Objective.

To learn how to perform a restore, see Restoring a DynamoDB table from a backup.

You can use IAM policies for access control. For more information, see Using IAM with DynamoDB backup and restore.

All backup and restore console and API actions are captured and recorded in AWS CloudTrail for logging, continuous monitoring, and auditing.

Creating backups of DynamoDB tables with AWS Backup

This section describes how to turn on AWS Backup to create on-demand and scheduled backups from your DynamoDB tables.

Topics

- Turning on AWS Backup features
- On-demand backups
- Scheduled backups

Turning on AWS Backup features

You must turn on AWS Backup to use it with DynamoDB.

To turn on AWS Backup, go through the following steps:

1. Sign in to the AWS Management Console and open the DynamoDB console at https://console.aws.amazon.com/dynamodb/.
2. In the navigation pane on the left side of the console, choose **Backups**.
3. In the Backup Settings window, choose **Turn on**.
4. A confirmation screen will appear. Choose **Turn on features**.

AWS Backup features are now available for your DynamoDB tables.

If you choose to turn off AWS Backup features after they've been turned on, follow these steps:

1. Sign in to the AWS Management Console and open the DynamoDB console at https://console.aws.amazon.com/dynamodb/.

2. In the navigation pane on the left side of the console, choose **Backups**.
3. In the Backup Settings window, choose **Turn off**.
4. A confirmation screen will appear. Choose **Turn off features**.

If you can't turn the AWS Backup features on or off, your AWS admin may need to perform those actions.

On-demand backups

To create an on-demand backup of a DynamoDB table, follow these steps:

1. Sign in to the AWS Management Console and open the DynamoDB console at https://console.aws.amazon.com/dynamodb/.
2. In the navigation pane on the left side of the console, choose **Backups**.
3. Choose **Create backup**.
4. From the dropdown menu that appears, choose **Create an on-demand backup**.
5. To create a backup managed by AWS Backup with warm storage and other basic features, choose **Default Settings.** To create a backup that can be transitioned to cold storage, or to create a backup with DynamoDB features instead of AWS Backup, choose **Customize settings**.

   If you want to create this backup with previous DynamoDB features instead, choose **Customize settings** and then choose **Backup with DynamoDB**.
6. When you have completed the settings, choose **Create backup.**

Scheduled backups

To schedule a backup, follow these steps.

1. Sign in to the AWS Management Console and open the DynamoDB console at https://console.aws.amazon.com/dynamodb/.
2. In the navigation pane on the left side of the console, choose **Backups**.
3. From the dropdown menu that appears, choose **Schedule backups with AWS Backup**.
4. You will be taken to AWS Backup to create a backup plan.

Copying a backup of a DynamoDB table with AWS Backup

You can make a copy of a current backup. You can copy backups to multiple AWS accounts or AWS Regions on demand or automatically as part of a scheduled backup plan. You can also automate a sequence of cross-account and cross-Region copies for Amazon DynamoDB Encryption Client.

Cross-Region replication is especially valuable if you have business continuity or compliance requirements to store backups a minimum distance away from your production data.

Cross-account backups are useful for securely copying your backups to one or more AWS accounts in your organization for operational or security reasons. If your original backup is inadvertently deleted, you can copy the backup from its destination account to its source account, and then start the restore. Before you can do this, you must have two accounts that belong to the same organization in the Organizations service.

Copies inherit the source backup's configuration unless you specify otherwise, with one exception: if you specify that your new copy "Never" expire. With this setting, the new copy still inherits its source expiration date. If you want your new backup copy to be permanent, either set your source backups to never expire, or specify your new copy to expire 100 years after its creation.

> **Note**
>
> If you're copying to another account, you must first have permission from that account.

To copy a backup, do the following:

1. Sign in to the AWS Management Console and open the DynamoDB console at https://console.aws.amazon.com/dynamodb/.
2. In the navigation pane on the left side of the console, choose **Backups**.
3. Select the check box next to the backup you want to copy.
   - If the backup you want to copy is grayed out, you must enable advanced features with AWS Backup. Then create a new backup. You can now copy this new backup to other Regions and accounts, and copy any other new backups going forward.
4. Choose **Copy.**
5. If you want to copy the backup to another account or Region, select the check box next to **Copy the recovery point to another destination**. Then select whether you will to copy to another Region in your account, or to a different account in a different Region.

> **Note**
>
> To restore a backup to another Region or account, you must first copy the backup to that Region or account.

6. Select the desired vault the file will be copied into. You can also create a new backup vault if desired.
7. Choose **Copy backup**.

Restoring a backup of a DynamoDB table from AWS Backup

This section describes how to restore a backup of a DynamoDB table from AWS Backup.

Topics

- Restoring a DynamoDB table from AWS Backup
- Restoring a DynamoDB table to another Region or account

Restoring a DynamoDB table from AWS Backup

To restore your DynamoDB tables from AWS Backup, follow these steps:

1. Sign in to the AWS Management Console and open the DynamoDB console at https://console.aws.amazon.com/dynamodb/
2. In the navigation pane on the left side of the console, choose **Tables.**
3. Choose the **Backups** tab.
4. Select the check box next to the previous backup that you want to restore from.
5. Choose **Restore**. You will be taken to the **Restore table from backup** screen.
6. Enter the name for the newly restored table, the encryption that this new table will have, the key you want the restore to be encryped with, and other options.
7. When you're finished, choose **Restore.**

Restoring a DynamoDB table to another Region or account

To restore a DynamoDB table to another Region or account, you will first need to copy the backup to that new Region or account. In order to copy to another account, that account must first grant

you permission. After you have copied your DynamoDB backup to the new Region or account, it can be restored with the process in the previous section.

Deleting a backup of a DynamoDB table with AWS Backup

This section describes how to delete a backup of a DynamoDB table with AWS Backup.

A DynamoDB backup created through AWS Backup features is stored in an AWS Backup vault.

In order to delete this kind of backup, do the following:

1. Sign in to the AWS Management Console and open the DynamoDB console at https://console.aws.amazon.com/dynamodb/.
2. In the navigation pane on the left side of the console, choose **Backups**.
3. On the screen that follows, choose **Continue to AWS Backup**.

   You will be taken to the AWS Backup console. To learn more on how to delete backups on the AWS Backup console, see Deleting backups.

   For more information about AWS Backup see Backup and recovery using AWS Backup in the *AWS Prescriptive Guidance*.

Usage note differences between on-demand backups managed by AWS Backup and DynamoDB

This section describes the technical differences between on-demand backups managed by AWS Backup and DynamoDB.

AWS Backup has some different workflows and behaviors than DynamoDB. These include:

Encryption - Backups created with the AWS Backup plan are stored in an encrypted vault with a key that is managed by the AWS Backup service. The vault has access control policies for additional security.

Backup ARN - The backup files created by AWS Backup will now have an AWS Backup ARN, which could impact the user permission model. Backup resource names (ARNs) will change from `arn:aws:dynamodb` to `arn:aws:backup`.

Deleting backups - Backups that are created with AWS Backup can only be deleted from the AWS Backup vault. You will not be able to delete AWS Backup files from the DynamoDB console.

Backup process - Unlike DynamoDB backups, backups made with AWS Backup are not instantaneous.

Billing - Backups of DynamoDB tables with AWS Backup features are billed from AWS Backup.

IAM roles - If you're managing access through IAM roles, you will also need to configure a new IAM role with these new permissions:

```
"dynamodb:StartAwsBackupJob",
"dynamodb:RestoreTableFromAwsBackup"
```

`dynamodb:StartAwsBackupJob` is needed for a successful backup with AWS Backup features, and `dynamodb:RestoreTableFromAwsBackup` is needed to restore from a backup made with AWS Backup features.

To see these permissions in a complete IAM policy, see Example 8 in Using IAM.

Code examples for DynamoDB using AWS SDKs

The following code examples show how to use DynamoDB with an AWS software development kit (SDK).

Basics are code examples that show you how to perform the essential operations within a service.

Actions are code excerpts from larger programs and must be run in context. While actions show you how to call individual service functions, you can see actions in context in their related scenarios.

Scenarios are code examples that show you how to accomplish specific tasks by calling multiple functions within a service or combined with other AWS services.

AWS community contributions are examples that were created and are maintained by multiple teams across AWS. To provide feedback, use the mechanism provided in the linked repositories.

For a complete list of AWS SDK developer guides and code examples, see Using DynamoDB with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Code examples

- Basic examples for DynamoDB using AWS SDKs
  - Hello DynamoDB
  - Learn the basics of DynamoDB with an AWS SDK
  - Actions for DynamoDB using AWS SDKs
    - Use BatchExecuteStatement with an AWS SDK
    - Use BatchGetItem with an AWS SDK or CLI
    - Use BatchWriteItem with an AWS SDK or CLI
    - Use CreateTable with an AWS SDK or CLI
    - Use DeleteItem with an AWS SDK or CLI
    - Use DeleteTable with an AWS SDK or CLI
    - Use DescribeTable with an AWS SDK or CLI
    - Use DescribeTimeToLive with an AWS SDK or CLI
    - Use ExecuteStatement with an AWS SDK
    - Use GetItem with an AWS SDK or CLI

    - Use ListTables with an AWS SDK or CLI
    - Use PutItem with an AWS SDK or CLI
    - Use Query with an AWS SDK or CLI
    - Use Scan with an AWS SDK or CLI
    - Use UpdateItem with an AWS SDK or CLI
    - Use UpdateTable with an AWS SDK or CLI
    - Use UpdateTimeToLive with an AWS SDK or CLI
- Scenarios for DynamoDB using AWS SDKs
    - Accelerate DynamoDB reads with DAX using an AWS SDK
    - Work with advanced DynamoDB Global Secondary Index scenarios using AWS Command Line Interface v2
    - Build an application to submit data to a DynamoDB table
    - Compare multiple values with a single attribute in DynamoDB with an AWS SDK
    - Conditionally update a DynamoDB item with a TTL using an AWS SDK
    - Connect to a local DynamoDB instance using an AWS SDK
    - Count expression operators in DynamoDB with an AWS SDK
    - Create an API Gateway REST API to track COVID-19 data
    - Create a messenger application with Step Functions
    - Create a photo asset management application that lets users manage photos using labels
    - Create a DynamoDB table with a Global Secondary Index using the AWS SDK
    - Create a DynamoDB table with warm throughput setting using an AWS SDK
    - Create a web application to track DynamoDB data
    - Create a websocket chat application with API Gateway
    - Create a DynamoDB item with a TTL using an AWS SDK
    - Create and manage DynamoDB global tables with Multi-Region Strong Consistency using an AWS SDK
    - Create and manage DynamoDB global tables demonstrating MREC using an AWS SDK
    - Delete DynamoDB data using PartiQL DELETE statements with an AWS SDK
    - Detect PPE in images with Amazon Rekognition using an AWS SDK
    - Insert DynamoDB data using PartiQL INSERT statements with an AWS SDK
    - Invoke a Lambda function from a browser

- Manage DynamoDB Global Secondary Indexes using AWS Command Line Interface v2
- Manage DynamoDB resource-based policies using AWS Command Line Interface v2
- Monitor performance of Amazon DynamoDB using an AWS SDK
- Perform advanced DynamoDB query operations using an AWS SDK
- Perform list operations in DynamoDB with an AWS SDK
- Perform map operations in DynamoDB with an AWS SDK
- Perform set operations in DynamoDB with an AWS SDK
- Query a DynamoDB table by using batches of PartiQL statements and an AWS SDK
- Query a DynamoDB table using PartiQL and an AWS SDK
- Query a DynamoDB table using a Global Secondary Index with an AWS SDK
- Query a DynamoDB table using a begins_with condition with an AWS SDK
- Query a DynamoDB table using a date range in the sort key with an AWS SDK
- Query a DynamoDB table with a complex filter expression with an AWS SDK
- Query a DynamoDB table with a dynamic filter expression with an AWS SDK
- Query a DynamoDB table with a filter expression and limit with an AWS SDK
- Query a DynamoDB table with nested attributes using an AWS SDK
- Query a DynamoDB table with pagination using an AWS SDK
- Query a DynamoDB table with strongly consistent reads using an AWS SDK
- Query DynamoDB data using PartiQL SELECT statements with an AWS SDK
- Query a DynamoDB table for TTL items using an AWS SDK
- Query DynamoDB tables using date and time patterns with an AWS SDK
- Save EXIF and other image information using an AWS SDK
- Set up Attribute-Based Access Control for DynamoDB using AWS Command Line Interface v2
- Understand update expression order in DynamoDB with an AWS SDK
- Update a DynamoDB table setting with warm throughput using an AWS SDK
- Update a DynamoDB item with a TTL using an AWS SDK
- Update DynamoDB data using PartiQL UPDATE statements with an AWS SDK
- Use API Gateway to invoke a Lambda function
- Use Step Functions to invoke Lambda functions
- Use a document model for DynamoDB using an AWS SDK

- Use a high-level object persistence model for DynamoDB using an AWS SDK
- Use atomic counter operations in DynamoDB with an AWS SDK
- Use conditional operations in DynamoDB with an AWS SDK
- Use expression attribute names in DynamoDB with an AWS SDK
- Use scheduled events to invoke a Lambda function
- Work with DynamoDB Local Secondary Indexes using AWS Command Line Interface v2
- Work with DynamoDB Streams and Time-to-Live using AWS Command Line Interface v2
- Work with DynamoDB global tables and multi-Region replication with eventual consistency (MREC) using AWS Command Line Interface v2
- Work with DynamoDB resource tagging using AWS Command Line Interface v2
- Work with DynamoDB table encryption using AWS Command Line Interface v2
- Serverless examples for DynamoDB
  - Invoke a Lambda function from a DynamoDB trigger
  - Reporting batch item failures for Lambda functions with a DynamoDB trigger
- AWS community contributions for DynamoDB
  - Build and test a serverless application

Basic examples for DynamoDB using AWS SDKs

The following code examples show how to use the basics of Amazon DynamoDB with AWS SDKs.

Examples

- Hello DynamoDB
- Learn the basics of DynamoDB with an AWS SDK
- Actions for DynamoDB using AWS SDKs
  - Use BatchExecuteStatement with an AWS SDK
  - Use BatchGetItem with an AWS SDK or CLI
  - Use BatchWriteItem with an AWS SDK or CLI
  - Use CreateTable with an AWS SDK or CLI
  - Use DeleteItem with an AWS SDK or CLI
  - Use DeleteTable with an AWS SDK or CLI

- Use DescribeTable with an AWS SDK or CLI
- Use DescribeTimeToLive with an AWS SDK or CLI
- Use ExecuteStatement with an AWS SDK
- Use GetItem with an AWS SDK or CLI
- Use ListTables with an AWS SDK or CLI
- Use PutItem with an AWS SDK or CLI
- Use Query with an AWS SDK or CLI
- Use Scan with an AWS SDK or CLI
- Use UpdateItem with an AWS SDK or CLI
- Use UpdateTable with an AWS SDK or CLI
- Use UpdateTimeToLive with an AWS SDK or CLI

Hello DynamoDB

The following code examples show how to get started using DynamoDB.

.NET

SDK for .NET (v4)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
using Amazon.DynamoDBv2;
using Amazon.DynamoDBv2.Model;
using Microsoft.Extensions.DependencyInjection;

namespace DynamoDBActions;

/// <summary>
/// A simple example that demonstrates basic DynamoDB operations.
/// </summary>
public class HelloDynamoDB
```

```
{
    /// <summary>
    /// HelloDynamoDB lists the existing DynamoDB tables for the default user.
    /// </summary>
    /// <param name="args">Command line arguments</param>
    /// <returns>Async task.</returns>
    static async Task Main(string[] args)
    {
        // Set up dependency injection for Amazon DynamoDB.
        using var host =
 Microsoft.Extensions.Hosting.Host.CreateDefaultBuilder(args)
            .ConfigureServices((_, services) =>
                services.AddAWSService<IAmazonDynamoDB>()
            )
            .Build();

        // Now the client is available for injection.
        var dynamoDbClient = host.Services.GetRequiredService<IAmazonDynamoDB>();

        try
        {
            var request = new ListTablesRequest();
            var tableNames = new List<string>();

            var paginatorForTables =
 dynamoDbClient.Paginators.ListTables(request);

            await foreach (var tableName in paginatorForTables.TableNames)
            {
                tableNames.Add(tableName);
            }

            Console.WriteLine("Welcome to the DynamoDB Hello Service example. " +
                              "\nLet's list your DynamoDB tables:");
            tableNames.ForEach(table =>
            {
                Console.WriteLine($"Table: {table}");
            });
        }
        catch (AmazonDynamoDBException ex)
        {
            Console.WriteLine($"An Amazon DynamoDB service error occurred while
 listing tables. {ex.Message}");
        }
```

```
        catch (Exception ex)
        {
            Console.WriteLine($"An error occurred while listing tables.
 {ex.Message}");
        }
    }
}
```

- For API details, see ListTables in *AWS SDK for .NET API Reference*.

C++

SDK for C++

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Code for the CMakeLists.txt CMake file.

```
# Set the minimum required version of CMake for this project.
cmake_minimum_required(VERSION 3.13)

# Set the AWS service components used by this project.
set(SERVICE_COMPONENTS dynamodb)

# Set this project's name.
project("hello_dynamodb")

# Set the C++ standard to use to build this target.
# At least C++ 11 is required for the AWS SDK for C++.
set(CMAKE_CXX_STANDARD 11)

# Use the MSVC variable to determine if this is a Windows build.
set(WINDOWS_BUILD ${MSVC})

if (WINDOWS_BUILD) # Set the location where CMake can find the installed
 libraries for the AWS SDK.
```

```
    string(REPLACE ";" "/aws-cpp-sdk-all;" SYSTEM_MODULE_PATH
 "${CMAKE_SYSTEM_PREFIX_PATH}/aws-cpp-sdk-all")
    list(APPEND CMAKE_PREFIX_PATH ${SYSTEM_MODULE_PATH})
endif ()

# Find the AWS SDK for C++ package.
find_package(AWSSDK REQUIRED COMPONENTS ${SERVICE_COMPONENTS})

if (WINDOWS_BUILD AND AWSSDK_INSTALL_AS_SHARED_LIBS)
     # Copy relevant AWS SDK for C++ libraries into the current binary directory
 for running and debugging.

     # set(BIN_SUB_DIR "/Debug") # if you are building from the command line you
 may need to uncomment this
                                    # and set the proper subdirectory to the
 executables' location.

     AWSSDK_CPY_DYN_LIBS(SERVICE_COMPONENTS ""
 ${CMAKE_CURRENT_BINARY_DIR}${BIN_SUB_DIR})
endif ()

add_executable(${PROJECT_NAME}
        hello_dynamodb.cpp)

target_link_libraries(${PROJECT_NAME}
        ${AWSSDK_LINK_LIBRARIES})
```

Code for the hello_dynamodb.cpp source file.

```
#include <aws/core/Aws.h>
#include <aws/dynamodb/DynamoDBClient.h>
#include <aws/dynamodb/model/ListTablesRequest.h>
#include <iostream>

/*
 *  A "Hello DynamoDB" starter application which initializes an Amazon DynamoDB
 (DynamoDB) client and lists the
 *  DynamoDB tables.
 *
 *  main function
 *
 *  Usage: 'hello_dynamodb'
```

```
 *
 */

int main(int argc, char **argv) {
    Aws::SDKOptions options;
    // Optionally change the log level for debugging.
//   options.loggingOptions.logLevel = Utils::Logging::LogLevel::Debug;
    Aws::InitAPI(options); // Should only be called once.

    int result = 0;
    {
        Aws::Client::ClientConfiguration clientConfig;
        // Optional: Set to the AWS Region (overrides config file).
        // clientConfig.region = "us-east-1";

        Aws::DynamoDB::DynamoDBClient dynamodbClient(clientConfig);
        Aws::DynamoDB::Model::ListTablesRequest listTablesRequest;
        listTablesRequest.SetLimit(50);
        do {
            const Aws::DynamoDB::Model::ListTablesOutcome &outcome =
 dynamodbClient.ListTables(
                    listTablesRequest);
            if (!outcome.IsSuccess()) {
                std::cout << "Error: " << outcome.GetError().GetMessage() <<
 std::endl;
                result = 1;
                break;
            }

            for (const auto &tableName: outcome.GetResult().GetTableNames()) {
                std::cout << tableName << std::endl;
            }

            listTablesRequest.SetExclusiveStartTableName(
                    outcome.GetResult().GetLastEvaluatedTableName());

        } while (!listTablesRequest.GetExclusiveStartTableName().empty());
    }


    Aws::ShutdownAPI(options); // Should only be called once.
    return result;
}
```

- For API details, see ListTables in *AWS SDK for C++ API Reference*.

Java

SDK for Java 2.x

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.ListTablesRequest;
import software.amazon.awssdk.services.dynamodb.model.ListTablesResponse;
import java.util.List;

/**
 * Before running this Java V2 code example, set up your development
 * environment, including your credentials.
 *
 * For more information, see the following documentation topic:
 *
 * https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-
started.html
 */
public class ListTables {
    public static void main(String[] args) {
        System.out.println("Listing your Amazon DynamoDB tables:\n");
        Region region = Region.US_EAST_1;
        DynamoDbClient ddb = DynamoDbClient.builder()
                .region(region)
                .build();
        listAllTables(ddb);
        ddb.close();
    }

    public static void listAllTables(DynamoDbClient ddb) {
```

```
        boolean moreTables = true;
        String lastName = null;

        while (moreTables) {
            try {
                ListTablesResponse response = null;
                if (lastName == null) {
                    ListTablesRequest request =
 ListTablesRequest.builder().build();
                    response = ddb.listTables(request);
                } else {
                    ListTablesRequest request = ListTablesRequest.builder()
                            .exclusiveStartTableName(lastName).build();
                    response = ddb.listTables(request);
                }

                List<String> tableNames = response.tableNames();
                if (tableNames.size() > 0) {
                    for (String curName : tableNames) {
                        System.out.format("* %s\n", curName);
                    }
                } else {
                    System.out.println("No tables found!");
                    System.exit(0);
                }

                lastName = response.lastEvaluatedTableName();
                if (lastName == null) {
                    moreTables = false;
                }

            } catch (DynamoDbException e) {
                System.err.println(e.getMessage());
                System.exit(1);
            }
        }
        System.out.println("\nDone!");
    }
}
```

- For API details, see ListTables in *AWS SDK for Java 2.x API Reference*.

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

For more details on working with DynamoDB in AWS SDK for JavaScript, see Programming DynamoDB with JavaScript.

```
import { ListTablesCommand, DynamoDBClient } from "@aws-sdk/client-dynamodb";

const client = new DynamoDBClient({});

export const main = async () => {
  const command = new ListTablesCommand({});

  const response = await client.send(command);
  console.log(response.TableNames.join("\n"));
  return response;
};
```

- For API details, see ListTables in *AWS SDK for JavaScript API Reference*.

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import boto3

# Create a DynamoDB client using the default credentials and region
dynamodb = boto3.client("dynamodb")

# Initialize a paginator for the list_tables operation
paginator = dynamodb.get_paginator("list_tables")

# Create a PageIterator from the paginator
page_iterator = paginator.paginate(Limit=10)

# List the tables in the current AWS account
print("Here are the DynamoDB tables in your account:")

# Use pagination to list all tables
table_names = []

for page in page_iterator:
    for table_name in page.get("TableNames", []):
        print(f"- {table_name}")
        table_names.append(table_name)

if not table_names:
    print("You don't have any DynamoDB tables in your account.")
else:
    print(f"\nFound {len(table_names)} tables.")
```

- For API details, see ListTables in *AWS SDK for Python (Boto3) API Reference*.

Ruby

SDK for Ruby

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
require 'aws-sdk-dynamodb'
require 'logger'

# DynamoDBManager is a class responsible for managing DynamoDB operations
# such as listing all tables in the current AWS account.
class DynamoDBManager
  def initialize(client)
    @client = client
    @logger = Logger.new($stdout)
  end

  # Lists and prints all DynamoDB tables in the current AWS account.
  def list_tables
    @logger.info('Here are the DynamoDB tables in your account:')

    paginator = @client.list_tables(limit: 10)
    table_names = []

    paginator.each_page do |page|
      page.table_names.each do |table_name|
        @logger.info("- #{table_name}")
        table_names << table_name
      end
    end

    if table_names.empty?
      @logger.info("You don't have any DynamoDB tables in your account.")
    else
      @logger.info("\nFound #{table_names.length} tables.")
    end
  end
end

if $PROGRAM_NAME == __FILE__
  dynamodb_client = Aws::DynamoDB::Client.new
  manager = DynamoDBManager.new(dynamodb_client)
  manager.list_tables
end
```

- For API details, see ListTables in *AWS SDK for Ruby API Reference*.

For a complete list of AWS SDK developer guides and code examples, see Using DynamoDB with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Learn the basics of DynamoDB with an AWS SDK

The following code examples show how to:

- Create a table that can hold movie data.
- Put, get, and update a single movie in the table.
- Write movie data to the table from a sample JSON file.
- Query for movies that were released in a given year.
- Scan for movies that were released in a range of years.
- Delete a movie from the table, then delete the table.

.NET

SDK for .NET (v4)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
/// <summary>
/// This example application performs the following basic Amazon DynamoDB
/// functions:
///     CreateTableAsync
///     PutItemAsync
///     UpdateItemAsync
///     BatchWriteItemAsync
///     GetItemAsync
///     DeleteItemAsync
///     Query
///     Scan
///     DeleteItemAsync.
/// </summary>
```

```
public class DynamoDbBasics
{
    public static bool IsInteractive = true;

    // Separator for the console display.
    private static readonly string SepBar = new string('-', 80);

    /// <summary>
    /// The main entry point for the DynamoDB Basics example application.
    /// </summary>
    /// <param name="args">Command line arguments.</param>
    /// <returns>A task representing the asynchronous operation.</returns>
    public static async Task Main(string[] args)
    {
        // Set up dependency injection for Amazon DynamoDB.
        using var host =
 Microsoft.Extensions.Hosting.Host.CreateDefaultBuilder(args)
            .ConfigureServices((_, services) =>
                services.AddAWSService<IAmazonDynamoDB>()
                    .AddTransient<DynamoDbWrapper>())
            .Build();

        // Now the wrapper is available for injection.
        var dynamoDbWrapper =
 host.Services.GetRequiredService<DynamoDbWrapper>();

        var tableName = "movie_table";

        var movieFileName = @"movies.json";

        DisplayInstructions();

        // Create a new table and wait for it to be active.
        Console.WriteLine($"Creating the new table: {tableName}");

        var success = await dynamoDbWrapper.CreateMovieTableAsync(tableName);

        Console.WriteLine(success
            ? $"\nTable: {tableName} successfully created."
            : $"\nCould not create {tableName}.");

        WaitForEnter();

        // Add a single new movie to the table.
```

```
        var newMovie = new Movie
        {
            Year = 2021,
            Title = "Spider-Man: No Way Home",
        };

        success = await dynamoDbWrapper.PutItemAsync(newMovie, tableName);
        if (success)
        {
            Console.WriteLine($"Added {newMovie.Title} to the table.");
        }
        else
        {
            Console.WriteLine("Could not add movie to table.");
        }

        WaitForEnter();

        // Update the new movie by adding a plot and rank.
        var newInfo = new MovieInfo
        {
            Plot = "With Spider-Man's identity now revealed, Peter asks" +
                   "Doctor Strange for help. When a spell goes wrong, dangerous"
+
                   "foes from other worlds start to appear, forcing Peter to" +
                   "discover what it truly means to be Spider-Man.",
            Rank = 9,
        };

        success = await dynamoDbWrapper.UpdateItemAsync(newMovie, newInfo,
tableName);
        if (success)
        {
            Console.WriteLine($"Successfully updated the movie:
{newMovie.Title}");
        }
        else
        {
            Console.WriteLine("Could not update the movie.");
        }

        WaitForEnter();

        // Add a batch of movies to the DynamoDB table from a list of
```

```
        // movies in a JSON file.
        var itemCount = await dynamoDbWrapper.BatchWriteItemsAsync(movieFileName,
tableName);
        Console.WriteLine($"Added {itemCount} movies to the table.");

        WaitForEnter();

        // Get a movie by key. (partition + sort)
        var lookupMovie = new Movie
        {
            Title = "Jurassic Park",
            Year = 1993,
        };

        Console.WriteLine("Looking for the movie \"Jurassic Park\".");
        var item = await dynamoDbWrapper.GetItemAsync(lookupMovie, tableName);
        if (item?.Count > 0)
        {
            dynamoDbWrapper.DisplayItem(item);
        }
        else
        {
            Console.WriteLine($"Couldn't find {lookupMovie.Title}");
        }

        WaitForEnter();

        // Delete a movie.
        var movieToDelete = new Movie
        {
            Title = "The Town",
            Year = 2010,
        };

        success = await dynamoDbWrapper.DeleteItemAsync(tableName,
movieToDelete);

        if (success)
        {
            Console.WriteLine($"Successfully deleted {movieToDelete.Title}.");
        }
        else
        {
            Console.WriteLine($"Could not delete {movieToDelete.Title}.");
```

```
        }

        WaitForEnter();

        // Use Query to find all the movies released in 2010.
        int findYear = 2010;
        Console.WriteLine($"Movies released in {findYear}");
        var queryCount = await dynamoDbWrapper.QueryMoviesAsync(tableName,
findYear);
        Console.WriteLine($"Found {queryCount} movies released in {findYear}");

        WaitForEnter();

        // Use Scan to get a list of movies from 2001 to 2011.
        int startYear = 2001;
        int endYear = 2011;
        var scanCount = await dynamoDbWrapper.ScanTableAsync(tableName,
startYear, endYear);
        Console.WriteLine($"Found {scanCount} movies released between {startYear}
and {endYear}");

        WaitForEnter();

        // Delete the table.
        success = await dynamoDbWrapper.DeleteTableAsync(tableName);

        if (success)
        {
            Console.WriteLine($"Successfully deleted {tableName}");
        }
        else
        {
            Console.WriteLine($"Could not delete {tableName}");
        }

        Console.WriteLine("The DynamoDB Basics example application is
complete.");

        WaitForEnter();
    }

    /// <summary>
    /// Displays the description of the application on the console.
    /// </summary>
```

```
    private static void DisplayInstructions()
    {
        if (!IsInteractive)
        {
            return;
        }

        Console.Clear();
        Console.WriteLine();
        Console.Write(new string(' ', 28));
        Console.WriteLine("DynamoDB Basics Example");
        Console.WriteLine(SepBar);
        Console.WriteLine("This demo application shows the basics of using
DynamoDB with the AWS SDK.");
        Console.WriteLine(SepBar);
        Console.WriteLine("The application does the following:");
        Console.WriteLine("\t1. Creates a table with partition: year and
sort:title.");
        Console.WriteLine("\t2. Adds a single movie to the table.");
        Console.WriteLine("\t3. Adds movies to the table from moviedata.json.");
        Console.WriteLine("\t4. Updates the rating and plot of the movie that was
just added.");
        Console.WriteLine("\t5. Gets a movie using its key (partition + sort).");
        Console.WriteLine("\t6. Deletes a movie.");
        Console.WriteLine("\t7. Uses QueryAsync to return all movies released in
a given year.");
        Console.WriteLine("\t8. Uses ScanAsync to return all movies released
within a range of years.");
        Console.WriteLine("\t9. Finally, it deletes the table that was just
created.");
        WaitForEnter();
    }

    /// <summary>
    /// Simple method to wait for the Enter key to be pressed.
    /// </summary>
    private static void WaitForEnter()
    {
        if (IsInteractive)
        {
            Console.WriteLine("\nPress <Enter> to continue.");
            Console.WriteLine(SepBar);
            _ = Console.ReadLine();
        }
```

```
    }
}
```

Use the injected client for table operations.

```
using System.Text.Json;
using Amazon.DynamoDBv2;
using Amazon.DynamoDBv2.DataModel;
using Amazon.DynamoDBv2.DocumentModel;
using Amazon.DynamoDBv2.Model;

namespace DynamoDBActions;

/// <summary>
/// Methods of this class perform Amazon DynamoDB operations.
/// </summary>
public class DynamoDbWrapper
{
    private readonly IAmazonDynamoDB _amazonDynamoDB;

    /// <summary>
    /// Constructor for the DynamoDbWrapper class.
    /// </summary>
    /// <param name="amazonDynamoDB">The injected DynamoDB client.</param>
    public DynamoDbWrapper(IAmazonDynamoDB amazonDynamoDB)
    {
        _amazonDynamoDB = amazonDynamoDB;
    }
```

Creates a table to contain movie data.

```
    /// <summary>
    /// Creates a new Amazon DynamoDB table and then waits for the new
    /// table to become active.
    /// </summary>
    /// <param name="tableName">The name of the table to create.</param>
    /// <returns>A Boolean value indicating the success of the operation.</
returns>
    public async Task<bool> CreateMovieTableAsync(string tableName)
```

```
    {
        try
        {
            var response = await _amazonDynamoDB.CreateTableAsync(new
CreateTableRequest
            {
                TableName = tableName,
                AttributeDefinitions = new List<AttributeDefinition>()
                {
                    new AttributeDefinition
                    {
                        AttributeName = "title",
                        AttributeType = ScalarAttributeType.S,
                    },
                    new AttributeDefinition
                    {
                        AttributeName = "year",
                        AttributeType = ScalarAttributeType.N,
                    },
                },
                KeySchema = new List<KeySchemaElement>()
                {
                    new KeySchemaElement
                    {
                        AttributeName = "year",
                        KeyType = KeyType.HASH,
                    },
                    new KeySchemaElement
                    {
                        AttributeName = "title",
                        KeyType = KeyType.RANGE,
                    },
                },
                BillingMode = BillingMode.PAY_PER_REQUEST,
            });

            // Wait until the table is ACTIVE and then report success.
            Console.Write("Waiting for table to become active...");

            var request = new DescribeTableRequest
            {
                TableName = response.TableDescription.TableName,
            };
```

```
            TableStatus status;

            int sleepDuration = 2000;

            do
            {
                Thread.Sleep(sleepDuration);

                var describeTableResponse = await
_amazonDynamoDB.DescribeTableAsync(request);
                status = describeTableResponse.Table.TableStatus;

                Console.Write(".");
            }
            while (status != "ACTIVE");

            return status == TableStatus.ACTIVE;
        }
        catch (ResourceInUseException ex)
        {
            Console.WriteLine($"Table {tableName} already exists. {ex.Message}");
            throw;
        }
        catch (AmazonDynamoDBException ex)
        {
            Console.WriteLine($"An Amazon DynamoDB error occurred while creating
table {tableName}. {ex.Message}");
            throw;
        }
        catch (Exception ex)
        {
            Console.WriteLine($"An error occurred while creating table
{tableName}. {ex.Message}");
            throw;
        }
    }
```

Adds a single movie to the table.

```
    /// <summary>
```

```
    /// Adds a new item to the table.
    /// </summary>
    /// <param name="newMovie">A Movie object containing informtation for
    /// the movie to add to the table.</param>
    /// <param name="tableName">The name of the table where the item will be
 added.</param>
    /// <returns>A Boolean value that indicates the results of adding the item.</
returns>
    public async Task<bool> PutItemAsync(Movie newMovie, string tableName)
    {
        try
        {
            var item = new Dictionary<string, AttributeValue>
            {
                ["title"] = new AttributeValue { S = newMovie.Title },
                ["year"] = new AttributeValue { N = newMovie.Year.ToString() },
            };

            var request = new PutItemRequest
            {
                TableName = tableName,
                Item = item,
            };

            await _amazonDynamoDB.PutItemAsync(request);
            return true;
        }
        catch (ResourceNotFoundException ex)
        {
            Console.WriteLine($"Table {tableName} was not found. {ex.Message}");
            return false;
        }
        catch (AmazonDynamoDBException ex)
        {
            Console.WriteLine($"An Amazon DynamoDB error occurred while putting
 item. {ex.Message}");
            throw;
        }
        catch (Exception ex)
        {
            Console.WriteLine($"An error occurred while putting item.
 {ex.Message}");
            throw;
        }
```

```
    }
```

Updates a single item in a table.

```
    /// <summary>
    /// Updates an existing item in the movies table.
    /// </summary>
    /// <param name="newMovie">A Movie object containing information for
    /// the movie to update.</param>
    /// <param name="newInfo">A MovieInfo object that contains the
    /// information that will be changed.</param>
    /// <param name="tableName">The name of the table that contains the movie.</
param>
    /// <returns>A Boolean value that indicates the success of the operation.</
returns>
    public async Task<bool> UpdateItemAsync(
        Movie newMovie,
        MovieInfo newInfo,
        string tableName)
    {
        try
        {
            var key = new Dictionary<string, AttributeValue>
            {
                ["title"] = new AttributeValue { S = newMovie.Title },
                ["year"] = new AttributeValue { N = newMovie.Year.ToString() },
            };
            var updates = new Dictionary<string, AttributeValueUpdate>
            {
                ["info.plot"] = new AttributeValueUpdate
                {
                    Action = AttributeAction.PUT,
                    Value = new AttributeValue { S = newInfo.Plot },
                },

                ["info.rating"] = new AttributeValueUpdate
                {
                    Action = AttributeAction.PUT,
                    Value = new AttributeValue { N = newInfo.Rank.ToString() },
                },
```

```
            };

            var request = new UpdateItemRequest
            {
                AttributeUpdates = updates,
                Key = key,
                TableName = tableName,
            };

            await _amazonDynamoDB.UpdateItemAsync(request);
            return true;
        }
        catch (ResourceNotFoundException ex)
        {
            Console.WriteLine($"Table {tableName} or item was not found.
{ex.Message}");
            return false;
        }
        catch (AmazonDynamoDBException ex)
        {
            Console.WriteLine($"An Amazon DynamoDB error occurred while updating
item. {ex.Message}");
            throw;
        }
        catch (Exception ex)
        {
            Console.WriteLine($"An error occurred while updating item.
{ex.Message}");
            throw;
        }
    }
```

Retrieves a single item from the movie table.

```
    /// <summary>
    /// Gets information about an existing movie from the table.
    /// </summary>
    /// <param name="newMovie">A Movie object containing information about
    /// the movie to retrieve.</param>
```

```
    /// <param name="tableName">The name of the table containing the movie.</
param>
    /// <returns>A Dictionary object containing information about the item
    /// retrieved.</returns>
    public async Task<Dictionary<string, AttributeValue>> GetItemAsync(Movie
 newMovie, string tableName)
    {
        try
        {
            var key = new Dictionary<string, AttributeValue>
            {
                ["title"] = new AttributeValue { S = newMovie.Title },
                ["year"] = new AttributeValue { N = newMovie.Year.ToString() },
            };

            var request = new GetItemRequest
            {
                Key = key,
                TableName = tableName,
            };

            var response = await _amazonDynamoDB.GetItemAsync(request);
            return response.Item;
        }
        catch (ResourceNotFoundException ex)
        {
            Console.WriteLine($"Table {tableName} was not found. {ex.Message}");
            return new Dictionary<string, AttributeValue>();
        }
        catch (AmazonDynamoDBException ex)
        {
            Console.WriteLine($"An Amazon DynamoDB error occurred while getting
 item. {ex.Message}");
            throw;
        }
        catch (Exception ex)
        {
            Console.WriteLine($"An error occurred while getting item.
 {ex.Message}");
            throw;
        }
    }
```

Writes a batch of items to the movie table.

```
/// <summary>
/// Loads the contents of a JSON file into a list of movies to be
/// added to the DynamoDB table.
/// </summary>
/// <param name="movieFileName">The name of the JSON file.</param>
/// <returns>A generic list of movie objects.</returns>
public List<Movie> ImportMovies(string movieFileName)
{
    var moviesList = new List<Movie>();
    if (!File.Exists(movieFileName))
    {
        return moviesList;
    }

    using var sr = new StreamReader(movieFileName);
    string json = sr.ReadToEnd();
    var allMovies = JsonSerializer.Deserialize<List<Movie>>(
        json,
        new JsonSerializerOptions
        {
            PropertyNameCaseInsensitive = true
        });

    // Now return the first 250 entries.
    if (allMovies != null && allMovies.Any())
    {
        moviesList = allMovies.GetRange(0, 250);
    }
    return moviesList;
}

/// <summary>
/// Writes 250 items to the movie table.
/// </summary>
/// <param name="movieFileName">A string containing the full path to
/// the JSON file containing movie data.</param>
/// <param name="tableName">The name of the table to write items to.</param>
/// <returns>A long integer value representing the number of movies
```

```
    /// imported from the JSON file.</returns>
    public async Task<long> BatchWriteItemsAsync(
        string movieFileName, string tableName)
    {
        try
        {
            var movies = ImportMovies(movieFileName);
            if (!movies.Any())
            {
                Console.WriteLine("Couldn't find the JSON file with movie
data.");
                return 0;
            }

            var context = new DynamoDBContextBuilder()
                // Optional call to provide a specific instance of
IAmazonDynamoDB
                .WithDynamoDBClient(() => _amazonDynamoDB)
                .Build();

            var movieBatch = context.CreateBatchWrite<Movie>(
                new BatchWriteConfig()
                {
                    OverrideTableName = tableName
                });
            movieBatch.AddPutItems(movies);

            Console.WriteLine("Adding imported movies to the table.");
            await movieBatch.ExecuteAsync();

            return movies.Count;
        }
        catch (ResourceNotFoundException ex)
        {
            Console.WriteLine($"Table was not found during batch write operation.
{ex.Message}");
            throw;
        }
        catch (AmazonDynamoDBException ex)
        {
            Console.WriteLine($"An Amazon DynamoDB error occurred during batch
write operation. {ex.Message}");
            throw;
        }
```

```
        catch (Exception ex)
        {
            Console.WriteLine($"An error occurred during batch write operation.
{ex.Message}");
            throw;
        }
    }
```

Deletes a single item from the table.

```
    /// <summary>
    /// Deletes a single item from a DynamoDB table.
    /// </summary>
    /// <param name="tableName">The name of the table from which the item
    /// will be deleted.</param>
    /// <param name="movieToDelete">A movie object containing the title and
    /// year of the movie to delete.</param>
    /// <returns>A Boolean value indicating the success or failure of the
    /// delete operation.</returns>
    public async Task<bool> DeleteItemAsync(
        string tableName,
        Movie movieToDelete)
    {
        try
        {
            var key = new Dictionary<string, AttributeValue>
            {
                ["title"] = new AttributeValue { S = movieToDelete.Title },
                ["year"] = new AttributeValue { N =
movieToDelete.Year.ToString() },
            };

            var request = new DeleteItemRequest { TableName = tableName, Key =
key, };

            await _amazonDynamoDB.DeleteItemAsync(request);
            return true;
        }
        catch (ResourceNotFoundException ex)
        {
```

```
            Console.WriteLine($"Table {tableName} was not found. {ex.Message}");
            return false;
        }
        catch (AmazonDynamoDBException ex)
        {
            Console.WriteLine($"An Amazon DynamoDB error occurred while deleting
item. {ex.Message}");
            throw;
        }
        catch (Exception ex)
        {
            Console.WriteLine($"An error occurred while deleting item.
{ex.Message}");
            throw;
        }
    }
```

Queries the table for movies released in a particular year.

```
    /// <summary>
    /// Queries the table for movies released in a particular year and
    /// then displays the information for the movies returned.
    /// </summary>
    /// <param name="tableName">The name of the table to query.</param>
    /// <param name="year">The release year for which we want to
    /// view movies.</param>
    /// <returns>The number of movies that match the query.</returns>
    public async Task<int> QueryMoviesAsync(string tableName, int year)
    {
        try
        {
            var movieTable = new TableBuilder(_amazonDynamoDB, tableName)
                .AddHashKey("year", DynamoDBEntryType.Numeric)
                .AddRangeKey("title", DynamoDBEntryType.String)
                .Build();

            var filter = new QueryFilter("year", QueryOperator.Equal, year);

            Console.WriteLine("\nFind movies released in: {year}:");
```

```
            var config = new QueryOperationConfig()
            {
                Limit = 10, // 10 items per page.
                Select = SelectValues.SpecificAttributes,
                AttributesToGet = new List<string>
                {
                    "title",
                    "year",
                },
                ConsistentRead = true,
                Filter = filter,
            };

            // Value used to track how many movies match the
            // supplied criteria.
            var moviesFound = 0;

            var search = movieTable.Query(config);
            do
            {
                var movieList = await search.GetNextSetAsync();
                moviesFound += movieList.Count;

                foreach (var movie in movieList)
                {
                    DisplayDocument(movie);
                }
            }
            while (!search.IsDone);

            return moviesFound;
        }
        catch (ResourceNotFoundException ex)
        {
            Console.WriteLine($"Table {tableName} was not found. {ex.Message}");
            return 0;
        }
        catch (AmazonDynamoDBException ex)
        {
            Console.WriteLine($"An Amazon DynamoDB error occurred while querying
movies. {ex.Message}");
            throw;
        }
        catch (Exception ex)
```

```
        {
            Console.WriteLine($"An error occurred while querying movies.
 {ex.Message}");
            throw;
        }
    }
```

Scans the table for movies released in a range of years.

```
    /// <summary>
    /// Scans the table for movies released between the specified years.
    /// </summary>
    /// <param name="tableName">The name of the table to scan.</param>
    /// <param name="startYear">The starting year for the range.</param>
    /// <param name="endYear">The ending year for the range.</param>
    /// <returns>The number of movies found in the specified year range.</
returns>
    public async Task<int> ScanTableAsync(
        string tableName,
        int startYear,
        int endYear)
    {
        try
        {
            var request = new ScanRequest
            {
                TableName = tableName,
                ExpressionAttributeNames = new Dictionary<string, string>
                {
                    { "#yr", "year" },
                },
                ExpressionAttributeValues = new Dictionary<string,
 AttributeValue>
                {
                    { ":y_a", new AttributeValue { N = startYear.ToString() } },
                    { ":y_z", new AttributeValue { N = endYear.ToString() } },
                },
                FilterExpression = "#yr between :y_a and :y_z",
                ProjectionExpression = "#yr, title, info.actors[0],
 info.directors, info.running_time_secs",
```

```
                Limit = 10 // Set a limit to demonstrate using the
LastEvaluatedKey.
            };

            // Keep track of how many movies were found.
            int foundCount = 0;

            var response = new ScanResponse();
            do
            {
                response = await _amazonDynamoDB.ScanAsync(request);
                foundCount += response.Items.Count;
                response.Items.ForEach(i => DisplayItem(i));
                request.ExclusiveStartKey = response.LastEvaluatedKey;
            }
            while (response?.LastEvaluatedKey?.Count > 0);
            return foundCount;
        }
        catch (ResourceNotFoundException ex)
        {
            Console.WriteLine($"Table {tableName} was not found. {ex.Message}");
            return 0;
        }
        catch (AmazonDynamoDBException ex)
        {
            Console.WriteLine($"An Amazon DynamoDB error occurred while scanning
table. {ex.Message}");
            throw;
        }
        catch (Exception ex)
        {
            Console.WriteLine($"An error occurred while scanning table.
{ex.Message}");
            throw;
        }
    }
```

Deletes the movie table.

```
    /// <summary>
    /// Deletes a DynamoDB table.
```

```
    /// </summary>
    /// <param name="tableName">The name of the table to delete.</param>
    /// <returns>A Boolean value indicating the success of the operation.</
returns>
    public async Task<bool> DeleteTableAsync(string tableName)
    {
        try
        {
            var request = new DeleteTableRequest
            {
                TableName = tableName,
            };

            var response = await _amazonDynamoDB.DeleteTableAsync(request);

            Console.WriteLine($"Table {response.TableDescription.TableName}
 successfully deleted.");
            return true;

        }
        catch (ResourceNotFoundException ex)
        {
            Console.WriteLine($"Table {tableName} was not found and cannot be
 deleted. {ex.Message}");
            return false;
        }
        catch (AmazonDynamoDBException ex)
        {
            Console.WriteLine($"An Amazon DynamoDB error occurred while deleting
 table {tableName}. {ex.Message}");
            return false;
        }
        catch (Exception ex)
        {
            Console.WriteLine($"An error occurred while deleting table
 {tableName}. {ex.Message}");
            return false;
        }
    }
```

- For API details, see the following topics in *AWS SDK for .NET API Reference*.

- BatchWriteItem
- CreateTable
- DeleteItem
- DeleteTable
- DescribeTable
- GetItem
- PutItem
- Query
- Scan
- UpdateItem

Bash

AWS CLI with Bash script

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

The DynamoDB getting started scenario.

```
###############################################################################
# function dynamodb_getting_started_movies
#
# Scenario to create an Amazon DynamoDB table and perform a series of operations
 on the table.
#
# Returns:
#       0 - If successful.
#       1 - If an error occurred.
###############################################################################
function dynamodb_getting_started_movies() {

  source ./dynamodb_operations.sh

  key_schema_json_file="dynamodb_key_schema.json"
```

```
  attribute_definitions_json_file="dynamodb_attr_def.json"
  item_json_file="movie_item.json"
  key_json_file="movie_key.json"
  batch_json_file="batch.json"
  attribute_names_json_file="attribute_names.json"
  attributes_values_json_file="attribute_values.json"

  echo_repeat "*" 88
  echo
  echo "Welcome to the Amazon DynamoDB getting started demo."
  echo
  echo_repeat "*" 88
  echo

  local table_name
  echo -n "Enter a name for a new DynamoDB table: "
  get_input
  table_name=$get_input_result

  echo '[
  {"AttributeName": "year", "KeyType": "HASH"},
   {"AttributeName": "title", "KeyType": "RANGE"}
  ]' >"$key_schema_json_file"

  echo '[
  {"AttributeName": "year", "AttributeType": "N"},
   {"AttributeName": "title", "AttributeType": "S"}
  ]' >"$attribute_definitions_json_file"

  if dynamodb_create_table -n "$table_name" -a "$attribute_definitions_json_file"
\
    -k "$key_schema_json_file" 1>/dev/null; then
    echo "Created a DynamoDB table named $table_name"
  else
    errecho "The table failed to create. This demo will exit."
    clean_up
    return 1
  fi

  echo "Waiting for the table to become active...."

  if dynamodb_wait_table_active -n "$table_name"; then
    echo "The table is now active."
  else
```

```
    errecho "The table failed to become active. This demo will exit."
    cleanup "$table_name"
    return 1
  fi

  echo
  echo_repeat "*" 88
  echo

  echo -n "Enter the title of a movie you want to add to the table: "
  get_input
  local added_title
  added_title=$get_input_result

  local added_year
  get_int_input "What year was it released? "
  added_year=$get_input_result

  local rating
  get_float_input "On a scale of 1 - 10, how do you rate it? " "1" "10"
  rating=$get_input_result

  local plot
  echo -n "Summarize the plot for me: "
  get_input
  plot=$get_input_result

  echo '{
    "year": {"N" :"'"$added_year"'"},
    "title": {"S" :  "'"$added_title"'"},
    "info": {"M" : {"plot": {"S" : "'"$plot"'"}, "rating":
{"N" :"'"$rating"'"} } }
   }' >"$item_json_file"

  if dynamodb_put_item -n "$table_name" -i "$item_json_file"; then
    echo "The movie '$added_title' was successfully added to the table
'$table_name'."
  else
    errecho "Put item failed. This demo will exit."
    clean_up "$table_name"
    return 1
  fi

  echo
```

```
  echo_repeat "*" 88
  echo

  echo "Let's update your movie '$added_title'."
  get_float_input "You rated it $rating, what new rating would you give it? " "1"
"10"
  rating=$get_input_result

  echo -n "You summarized the plot as '$plot'."
  echo "What would you say now? "
  get_input
  plot=$get_input_result

  echo '{
    "year": {"N" :"'"$added_year"'"},
    "title": {"S" :  "'"$added_title"'"}
    }' >"$key_json_file"

  echo '{
    ":r": {"N" :"'"$rating"'"},
    ":p": {"S" : "'"$plot"'"}
   }' >"$item_json_file"

  local update_expression="SET info.rating = :r, info.plot = :p"

  if dynamodb_update_item -n "$table_name" -k "$key_json_file" -e
"$update_expression" -v "$item_json_file"; then
    echo "Updated '$added_title' with new attributes."
  else
    errecho "Update item failed. This demo will exit."
    clean_up "$table_name"
    return 1
  fi

  echo
  echo_repeat "*" 88
  echo

  echo "We will now use batch write to upload 150 movie entries into the table."

  local batch_json
  for batch_json in movie_files/movies_*.json; do
    echo "{ \"$table_name\" : $(<"$batch_json") }" >"$batch_json_file"
    if dynamodb_batch_write_item -i "$batch_json_file" 1>/dev/null; then
```

```
      echo "Entries in $batch_json added to table."
    else
      errecho "Batch write failed. This demo will exit."
      clean_up "$table_name"
      return 1
    fi
  done

  local title="The Lord of the Rings: The Fellowship of the Ring"
  local year="2001"

  if get_yes_no_input "Let's move on...do you want to get info about '$title'?
(y/n) "; then
    echo '{
  "year": {"N" :"'"$year"'"},
  "title": {"S" :  "'"$title"'"}
  }' >"$key_json_file"
    local info
    info=$(dynamodb_get_item -n "$table_name" -k "$key_json_file")

    # shellcheck disable=SC2181
    if [[ ${?} -ne 0 ]]; then
      errecho "Get item failed. This demo will exit."
      clean_up "$table_name"
      return 1
    fi

    echo "Here is what I found:"
    echo "$info"
  fi

  local ask_for_year=true
  while [[ "$ask_for_year" == true ]]; do
    echo "Let's get a list of movies released in a given year."
    get_int_input "Enter a year between 1972 and 2018: " "1972" "2018"
    year=$get_input_result
    echo '{
    "#n": "year"
    }' >"$attribute_names_json_file"

    echo '{
    ":v": {"N" :"'"$year"'"}
    }' >"$attributes_values_json_file"
```

```
    response=$(dynamodb_query -n "$table_name" -k "#n=:v" -a
"$attribute_names_json_file" -v "$attributes_values_json_file")

    # shellcheck disable=SC2181
    if [[ ${?} -ne 0 ]]; then
      errecho "Query table failed. This demo will exit."
      clean_up "$table_name"
      return 1
    fi

    echo "Here is what I found:"
    echo "$response"

    if ! get_yes_no_input "Try another year? (y/n) "; then
      ask_for_year=false
    fi
  done

  echo "Now let's scan for movies released in a range of years. Enter a year: "
  get_int_input "Enter a year between 1972 and 2018: " "1972" "2018"
  local start=$get_input_result

  get_int_input "Enter another year: " "1972" "2018"
  local end=$get_input_result

  echo '{
    "#n": "year"
    }' >"$attribute_names_json_file"

  echo '{
    ":v1": {"N" : "'"$start"'"},
    ":v2": {"N" : "'"$end"'"}
    }' >"$attributes_values_json_file"

  response=$(dynamodb_scan -n "$table_name" -f "#n BETWEEN :v1 AND :v2" -a
"$attribute_names_json_file" -v "$attributes_values_json_file")

  # shellcheck disable=SC2181
  if [[ ${?} -ne 0 ]]; then
    errecho "Scan table failed. This demo will exit."
    clean_up "$table_name"
    return 1
  fi
```

```
  echo "Here is what I found:"
  echo "$response"

  echo
  echo_repeat "*" 88
  echo

  echo "Let's remove your movie '$added_title' from the table."

  if get_yes_no_input "Do you want to remove '$added_title'? (y/n) "; then
    echo '{
  "year": {"N" :"'"$added_year"'"},
  "title": {"S" :  "'"$added_title"'"}
  }' >"$key_json_file"

    if ! dynamodb_delete_item -n "$table_name" -k "$key_json_file"; then
      errecho "Delete item failed. This demo will exit."
      clean_up "$table_name"
      return 1
    fi
  fi

  if get_yes_no_input "Do you want to delete the table '$table_name'? (y/n) ";
 then
    if ! clean_up "$table_name"; then
      return 1
    fi
  else
    if ! clean_up; then
      return 1
    fi
  fi

  return 0
}
```

The DynamoDB functions used in this scenario.

```
##############################################################################
# function dynamodb_create_table
#
# This function creates an Amazon DynamoDB table.
```

```
#
# Parameters:
#       -n table_name  -- The name of the table to create.
#       -a attribute_definitions -- JSON file path of a list of attributes and
 their types.
#       -k key_schema -- JSON file path of a list of attributes and their key
 types.
#
#  Returns:
#       0 - If successful.
#       1 - If it fails.
###############################################################################
function dynamodb_create_table() {
  local table_name attribute_definitions key_schema response
  local option OPTARG # Required to use getopts command in a function.

  ######################################
  # Function usage explanation
  ######################################
  function usage() {
    echo "function dynamodb_create_table"
    echo "Creates an Amazon DynamoDB table with on-demand billing."
    echo " -n table_name  -- The name of the table to create."
    echo " -a attribute_definitions -- JSON file path of a list of attributes and
 their types."
    echo " -k key_schema -- JSON file path of a list of attributes and their key
 types."
    echo ""
  }

  # Retrieve the calling parameters.
  while getopts "n:a:k:h" option; do
    case "${option}" in
      n) table_name="${OPTARG}" ;;
      a) attribute_definitions="${OPTARG}" ;;
      k) key_schema="${OPTARG}" ;;
      h)
        usage
        return 0
        ;;
      \?)
        echo "Invalid parameter"
        usage
        return 1
```

```
        ;;
    esac
  done
  export OPTIND=1

  if [[ -z "$table_name" ]]; then
    errecho "ERROR: You must provide a table name with the -n parameter."
    usage
    return 1
  fi

  if [[ -z "$attribute_definitions" ]]; then
    errecho "ERROR: You must provide an attribute definitions json file path the
-a parameter."
    usage
    return 1
  fi

  if [[ -z "$key_schema" ]]; then
    errecho "ERROR: You must provide a key schema json file path the -k
parameter."
    usage
    return 1
  fi

  iecho "Parameters:\n"
  iecho "    table_name:   $table_name"
  iecho "    attribute_definitions:   $attribute_definitions"
  iecho "    key_schema:   $key_schema"
  iecho ""

  response=$(aws dynamodb create-table \
    --table-name "$table_name" \
    --attribute-definitions file://"$attribute_definitions" \
    --billing-mode PAY_PER_REQUEST \
    --key-schema file://"$key_schema" )

  local error_code=${?}

  if [[ $error_code -ne 0 ]]; then
    aws_cli_error_log $error_code
    errecho "ERROR: AWS reports create-table operation failed.$response"
    return 1
  fi
```

```
  return 0
}

############################################################################
# function dynamodb_describe_table
#
# This function returns the status of a DynamoDB table.
#
# Parameters:
#       -n table_name  -- The name of the table.
#
#  Response:
#       - TableStatus:
#     And:
#       0 - Table is active.
#       1 - If it fails.
############################################################################
function dynamodb_describe_table {
  local table_name
  local option OPTARG # Required to use getopts command in a function.

  ######################################
  # Function usage explanation
  ######################################
  function usage() {
    echo "function dynamodb_describe_table"
    echo "Describe the status of a DynamoDB table."
    echo "  -n table_name  -- The name of the table."
    echo ""
  }

  # Retrieve the calling parameters.
  while getopts "n:h" option; do
    case "${option}" in
      n) table_name="${OPTARG}" ;;
      h)
        usage
        return 0
        ;;
      \?)
        echo "Invalid parameter"
        usage
        return 1
```

```
        ;;
    esac
  done
  export OPTIND=1

  if [[ -z "$table_name" ]]; then
    errecho "ERROR: You must provide a table name with the -n parameter."
    usage
    return 1
  fi

  local table_status
    table_status=$(
      aws dynamodb describe-table \
        --table-name "$table_name" \
        --output text \
        --query 'Table.TableStatus'
    )

   local error_code=${?}

    if [[ $error_code -ne 0 ]]; then
      aws_cli_error_log "$error_code"
      errecho "ERROR: AWS reports describe-table operation failed.$table_status"
      return 1
    fi

  echo "$table_status"

  return 0
}

###############################################################################
# function dynamodb_put_item
#
# This function puts an item into a DynamoDB table.
#
# Parameters:
#       -n table_name  -- The name of the table.
#       -i item  -- Path to json file containing the item values.
#
#  Returns:
#       0 - If successful.
#       1 - If it fails.
```

```
###############################################################################
function dynamodb_put_item() {
  local table_name item response
  local option OPTARG # Required to use getopts command in a function.

  ######################################
  # Function usage explanation
  ######################################
  function usage() {
    echo "function dynamodb_put_item"
    echo "Put an item into a DynamoDB table."
    echo " -n table_name  -- The name of the table."
    echo " -i item  -- Path to json file containing the item values."
    echo ""
  }

  while getopts "n:i:h" option; do
    case "${option}" in
      n) table_name="${OPTARG}" ;;
      i) item="${OPTARG}" ;;
      h)
        usage
        return 0
        ;;
      \?)
        echo "Invalid parameter"
        usage
        return 1
        ;;
    esac
  done
  export OPTIND=1

  if [[ -z "$table_name" ]]; then
    errecho "ERROR: You must provide a table name with the -n parameter."
    usage
    return 1
  fi

  if [[ -z "$item" ]]; then
    errecho "ERROR: You must provide an item with the -i parameter."
    usage
    return 1
  fi
```

```
  iecho "Parameters:\n"
  iecho "    table_name:   $table_name"
  iecho "    item:   $item"
  iecho ""
  iecho ""

  response=$(aws dynamodb put-item \
    --table-name "$table_name" \
    --item file://"$item")

  local error_code=${?}

  if [[ $error_code -ne 0 ]]; then
    aws_cli_error_log $error_code
    errecho "ERROR: AWS reports put-item operation failed.$response"
    return 1
  fi

  return 0

}

###############################################################################
# function dynamodb_update_item
#
# This function updates an item in a DynamoDB table.
#
#
# Parameters:
#       -n table_name  -- The name of the table.
#       -k keys  -- Path to json file containing the keys that identify the item
 to update.
#       -e update expression  -- An expression that defines one or more
 attributes to be updated.
#       -v values  -- Path to json file containing the update values.
#
#  Returns:
#       0 - If successful.
#       1 - If it fails.
###############################################################################
function dynamodb_update_item() {
  local table_name keys update_expression values response
  local option OPTARG # Required to use getopts command in a function.
```

```
 ######################################
 # Function usage explanation
 ######################################
 function usage() {
   echo "function dynamodb_update_item"
   echo "Update an item in a DynamoDB table."
   echo " -n table_name  -- The name of the table."
   echo " -k keys  -- Path to json file containing the keys that identify the
item to update."
   echo " -e update expression  -- An expression that defines one or more
attributes to be updated."
   echo " -v values  -- Path to json file containing the update values."
   echo ""
 }

 while getopts "n:k:e:v:h" option; do
   case "${option}" in
     n) table_name="${OPTARG}" ;;
     k) keys="${OPTARG}" ;;
     e) update_expression="${OPTARG}" ;;
     v) values="${OPTARG}" ;;
     h)
       usage
       return 0
       ;;
     \?)
       echo "Invalid parameter"
       usage
       return 1
       ;;
   esac
 done
 export OPTIND=1

 if [[ -z "$table_name" ]]; then
   errecho "ERROR: You must provide a table name with the -n parameter."
   usage
   return 1
 fi

 if [[ -z "$keys" ]]; then
   errecho "ERROR: You must provide a keys json file path the -k parameter."
   usage
```

```
    return 1
  fi
  if [[ -z "$update_expression" ]]; then
    errecho "ERROR: You must provide an update expression with the -e parameter."
    usage
    return 1
  fi

  if [[ -z "$values" ]]; then
    errecho "ERROR: You must provide a values json file path the -v parameter."
    usage
    return 1
  fi

  iecho "Parameters:\n"
  iecho "    table_name:   $table_name"
  iecho "    keys:   $keys"
  iecho "    update_expression:   $update_expression"
  iecho "    values:   $values"

  response=$(aws dynamodb update-item \
    --table-name "$table_name" \
    --key file://"$keys" \
    --update-expression "$update_expression" \
    --expression-attribute-values file://"$values")

  local error_code=${?}

  if [[ $error_code -ne 0 ]]; then
    aws_cli_error_log $error_code
    errecho "ERROR: AWS reports update-item operation failed.$response"
    return 1
  fi

  return 0

}

########################################################################################
# function dynamodb_batch_write_item
#
# This function writes a batch of items into a DynamoDB table.
#
# Parameters:
```

```
#       -i item  -- Path to json file containing the items to write.
#
#  Returns:
#       0 - If successful.
#       1 - If it fails.
###############################################################################
function dynamodb_batch_write_item() {
  local item response
  local option OPTARG # Required to use getopts command in a function.

  ######################################
  # Function usage explanation
  ######################################
  function usage() {
    echo "function dynamodb_batch_write_item"
    echo "Write a batch of items into a DynamoDB table."
    echo " -i item  -- Path to json file containing the items to write."
    echo ""
  }
  while getopts "i:h" option; do
    case "${option}" in
      i) item="${OPTARG}" ;;
      h)
        usage
        return 0
        ;;
      \?)
        echo "Invalid parameter"
        usage
        return 1
        ;;
    esac
  done
  export OPTIND=1

  if [[ -z "$item" ]]; then
    errecho "ERROR: You must provide an item with the -i parameter."
    usage
    return 1
  fi

  iecho "Parameters:\n"
  iecho "    table_name:   $table_name"
  iecho "    item:   $item"
```

```
  iecho ""

  response=$(aws dynamodb batch-write-item \
    --request-items file://"$item")

  local error_code=${?}

  if [[ $error_code -ne 0 ]]; then
    aws_cli_error_log $error_code
    errecho "ERROR: AWS reports batch-write-item operation failed.$response"
    return 1
  fi

  return 0
}

###############################################################################
# function dynamodb_get_item
#
# This function gets an item from a DynamoDB table.
#
# Parameters:
#       -n table_name  -- The name of the table.
#       -k keys  -- Path to json file containing the keys that identify the item
 to get.
#       [-q query]  -- Optional JMESPath query expression.
#
#  Returns:
#       The item as text output.
#  And:
#       0 - If successful.
#       1 - If it fails.
###############################################################################
function dynamodb_get_item() {
  local table_name keys query response
  local option OPTARG # Required to use getopts command in a function.

  # ######################################
  # Function usage explanation
  ######################################
  function usage() {
    echo "function dynamodb_get_item"
    echo "Get an item from a DynamoDB table."
    echo " -n table_name  -- The name of the table."
```

```
    echo " -k keys  -- Path to json file containing the keys that identify the
item to get."
    echo " [-q query]  -- Optional JMESPath query expression."
    echo ""
  }
  query=""
  while getopts "n:k:q:h" option; do
    case "${option}" in
      n) table_name="${OPTARG}" ;;
      k) keys="${OPTARG}" ;;
      q) query="${OPTARG}" ;;
      h)
        usage
        return 0
        ;;
      \?)
        echo "Invalid parameter"
        usage
        return 1
        ;;
    esac
  done
  export OPTIND=1

  if [[ -z "$table_name" ]]; then
    errecho "ERROR: You must provide a table name with the -n parameter."
    usage
    return 1
  fi

  if [[ -z "$keys" ]]; then
    errecho "ERROR: You must provide a keys json file path the -k parameter."
    usage
    return 1
  fi

  if [[ -n "$query" ]]; then
    response=$(aws dynamodb get-item \
      --table-name "$table_name" \
      --key file://"$keys" \
      --output text \
      --query "$query")
  else
    response=$(
```

```
      aws dynamodb get-item \
        --table-name "$table_name" \
        --key file://"$keys" \
        --output text
    )
  fi

  local error_code=${?}

  if [[ $error_code -ne 0 ]]; then
    aws_cli_error_log $error_code
    errecho "ERROR: AWS reports get-item operation failed.$response"
    return 1
  fi

  if [[ -n "$query" ]]; then
    echo "$response" | sed "/^\t/s/\t//1" # Remove initial tab that the JMSEPath
 query inserts on some strings.
  else
    echo "$response"
  fi

  return 0
}

###############################################################################
# function dynamodb_query
#
# This function queries a DynamoDB table.
#
# Parameters:
#       -n table_name  -- The name of the table.
#       -k key_condition_expression -- The key condition expression.
#       -a attribute_names -- Path to JSON file containing the attribute names.
#       -v attribute_values -- Path to JSON file containing the attribute values.
#       [-p projection_expression]  -- Optional projection expression.
#
#  Returns:
#       The items as json output.
#  And:
#       0 - If successful.
#       1 - If it fails.
###############################################################################
function dynamodb_query() {
```

```
 local table_name key_condition_expression attribute_names attribute_values
projection_expression response
 local option OPTARG # Required to use getopts command in a function.

 # ######################################
 # Function usage explanation
 ######################################
 function usage() {
   echo "function dynamodb_query"
   echo "Query a DynamoDB table."
   echo " -n table_name  -- The name of the table."
   echo " -k key_condition_expression -- The key condition expression."
   echo " -a attribute_names -- Path to JSON file containing the attribute
names."
   echo " -v attribute_values -- Path to JSON file containing the attribute
values."
   echo " [-p projection_expression]  -- Optional projection expression."
   echo ""
 }

 while getopts "n:k:a:v:p:h" option; do
   case "${option}" in
     n) table_name="${OPTARG}" ;;
     k) key_condition_expression="${OPTARG}" ;;
     a) attribute_names="${OPTARG}" ;;
     v) attribute_values="${OPTARG}" ;;
     p) projection_expression="${OPTARG}" ;;
     h)
       usage
       return 0
       ;;
     \?)
       echo "Invalid parameter"
       usage
       return 1
       ;;
   esac
 done
 export OPTIND=1

 if [[ -z "$table_name" ]]; then
   errecho "ERROR: You must provide a table name with the -n parameter."
   usage
   return 1
```

```
 fi

 if [[ -z "$key_condition_expression" ]]; then
   errecho "ERROR: You must provide a key condition expression with the -k
parameter."
   usage
   return 1
 fi

 if [[ -z "$attribute_names" ]]; then
   errecho "ERROR: You must provide a attribute names with the -a parameter."
   usage
   return 1
 fi

 if [[ -z "$attribute_values" ]]; then
   errecho "ERROR: You must provide a attribute values with the -v parameter."
   usage
   return 1
 fi

 if [[ -z "$projection_expression" ]]; then
   response=$(aws dynamodb query \
     --table-name "$table_name" \
     --key-condition-expression "$key_condition_expression" \
     --expression-attribute-names file://"$attribute_names" \
     --expression-attribute-values file://"$attribute_values")
 else
   response=$(aws dynamodb query \
     --table-name "$table_name" \
     --key-condition-expression "$key_condition_expression" \
     --expression-attribute-names file://"$attribute_names" \
     --expression-attribute-values file://"$attribute_values" \
     --projection-expression "$projection_expression")
 fi

 local error_code=${?}

 if [[ $error_code -ne 0 ]]; then
   aws_cli_error_log $error_code
   errecho "ERROR: AWS reports query operation failed.$response"
   return 1
 fi
```

```
  echo "$response"

  return 0
}

###############################################################################
# function dynamodb_scan
#
# This function scans a DynamoDB table.
#
# Parameters:
#       -n table_name  -- The name of the table.
#       -f filter_expression  -- The filter expression.
#       -a expression_attribute_names -- Path to JSON file containing the
 expression attribute names.
#       -v expression_attribute_values -- Path to JSON file containing the
 expression attribute values.
#       [-p projection_expression]  -- Optional projection expression.
#
#  Returns:
#       The items as json output.
#  And:
#       0 - If successful.
#       1 - If it fails.
###############################################################################
function dynamodb_scan() {
  local table_name filter_expression expression_attribute_names
 expression_attribute_values projection_expression response
  local option OPTARG # Required to use getopts command in a function.

  # ######################################
  # Function usage explanation
  ######################################
  function usage() {
    echo "function dynamodb_scan"
    echo "Scan a DynamoDB table."
    echo " -n table_name  -- The name of the table."
    echo " -f filter_expression  -- The filter expression."
    echo " -a expression_attribute_names -- Path to JSON file containing the
 expression attribute names."
    echo " -v expression_attribute_values -- Path to JSON file containing the
 expression attribute values."
    echo " [-p projection_expression]  -- Optional projection expression."
    echo ""
```

```
 }

 while getopts "n:f:a:v:p:h" option; do
   case "${option}" in
     n) table_name="${OPTARG}" ;;
     f) filter_expression="${OPTARG}" ;;
     a) expression_attribute_names="${OPTARG}" ;;
     v) expression_attribute_values="${OPTARG}" ;;
     p) projection_expression="${OPTARG}" ;;
     h)
       usage
       return 0
       ;;
     \?)
       echo "Invalid parameter"
       usage
       return 1
       ;;
   esac
 done
 export OPTIND=1

 if [[ -z "$table_name" ]]; then
   errecho "ERROR: You must provide a table name with the -n parameter."
   usage
   return 1
 fi

 if [[ -z "$filter_expression" ]]; then
   errecho "ERROR: You must provide a filter expression with the -f parameter."
   usage
   return 1
 fi

 if [[ -z "$expression_attribute_names" ]]; then
   errecho "ERROR: You must provide expression attribute names with the -a
parameter."
   usage
   return 1
 fi

 if [[ -z "$expression_attribute_values" ]]; then
   errecho "ERROR: You must provide expression attribute values with the -v
parameter."
```

```
    usage
    return 1
  fi

  if [[ -z "$projection_expression" ]]; then
    response=$(aws dynamodb scan \
      --table-name "$table_name" \
      --filter-expression "$filter_expression" \
      --expression-attribute-names file://"$expression_attribute_names" \
      --expression-attribute-values file://"$expression_attribute_values")
  else
    response=$(aws dynamodb scan \
      --table-name "$table_name" \
      --filter-expression "$filter_expression" \
      --expression-attribute-names file://"$expression_attribute_names" \
      --expression-attribute-values file://"$expression_attribute_values" \
      --projection-expression "$projection_expression")
  fi

  local error_code=${?}

  if [[ $error_code -ne 0 ]]; then
    aws_cli_error_log $error_code
    errecho "ERROR: AWS reports scan operation failed.$response"
    return 1
  fi

  echo "$response"

  return 0
}

###############################################################################
# function dynamodb_delete_item
#
# This function deletes an item from a DynamoDB table.
#
# Parameters:
#       -n table_name  -- The name of the table.
#       -k keys  -- Path to json file containing the keys that identify the item
 to delete.
#
#  Returns:
#       0 - If successful.
```

```
#       1 - If it fails.
###############################################################################
function dynamodb_delete_item() {
  local table_name keys response
  local option OPTARG # Required to use getopts command in a function.

  # ######################################
  # Function usage explanation
  ######################################
  function usage() {
    echo "function dynamodb_delete_item"
    echo "Delete an item from a DynamoDB table."
    echo " -n table_name  -- The name of the table."
    echo " -k keys  -- Path to json file containing the keys that identify the
 item to delete."
    echo ""
  }
  while getopts "n:k:h" option; do
    case "${option}" in
      n) table_name="${OPTARG}" ;;
      k) keys="${OPTARG}" ;;
      h)
        usage
        return 0
        ;;
      \?)
        echo "Invalid parameter"
        usage
        return 1
        ;;
    esac
  done
  export OPTIND=1

  if [[ -z "$table_name" ]]; then
    errecho "ERROR: You must provide a table name with the -n parameter."
    usage
    return 1
  fi

  if [[ -z "$keys" ]]; then
    errecho "ERROR: You must provide a keys json file path the -k parameter."
    usage
    return 1
```

```
  fi

  iecho "Parameters:\n"
  iecho "    table_name:   $table_name"
  iecho "    keys:   $keys"
  iecho ""

  response=$(aws dynamodb delete-item \
    --table-name "$table_name" \
    --key file://"$keys")

  local error_code=${?}

  if [[ $error_code -ne 0 ]]; then
    aws_cli_error_log $error_code
    errecho "ERROR: AWS reports delete-item operation failed.$response"
    return 1
  fi

  return 0

}

###############################################################################
# function dynamodb_delete_table
#
# This function deletes a DynamoDB table.
#
# Parameters:
#       -n table_name  -- The name of the table to delete.
#
#  Returns:
#       0 - If successful.
#       1 - If it fails.
###############################################################################
function dynamodb_delete_table() {
  local table_name response
  local option OPTARG # Required to use getopts command in a function.

  # bashsupport disable=BP5008
  function usage() {
    echo "function dynamodb_delete_table"
    echo "Deletes an Amazon DynamoDB table."
    echo " -n table_name  -- The name of the table to delete."
```

```
    echo ""
  }

  # Retrieve the calling parameters.
  while getopts "n:h" option; do
    case "${option}" in
      n) table_name="${OPTARG}" ;;
      h)
        usage
        return 0
        ;;
      \?)
        echo "Invalid parameter"
        usage
        return 1
        ;;
    esac
  done
  export OPTIND=1

  if [[ -z "$table_name" ]]; then
    errecho "ERROR: You must provide a table name with the -n parameter."
    usage
    return 1
  fi

  iecho "Parameters:\n"
  iecho "    table_name:   $table_name"
  iecho ""

  response=$(aws dynamodb delete-table \
    --table-name "$table_name")

  local error_code=${?}

  if [[ $error_code -ne 0 ]]; then
    aws_cli_error_log $error_code
    errecho "ERROR: AWS reports delete-table operation failed.$response"
    return 1
  fi

  return 0
}
```

The utility functions used in this scenario.

```
###############################################################################
# function iecho
#
# This function enables the script to display the specified text only if
# the global variable $VERBOSE is set to true.
###############################################################################
function iecho() {
  if [[ $VERBOSE == true ]]; then
    echo "$@"
  fi
}

###############################################################################
# function errecho
#
# This function outputs everything sent to it to STDERR (standard error output).
###############################################################################
function errecho() {
  printf "%s\n" "$*" 1>&2
}

###############################################################################
# function aws_cli_error_log()
#
# This function is used to log the error messages from the AWS CLI.
#
# See https://docs.aws.amazon.com/cli/latest/topic/return-codes.html#cli-aws-
help-return-codes.
#
# The function expects the following argument:
#         $1 - The error code returned by the AWS CLI.
#
#  Returns:
#          0: - Success.
#
###############################################################################
function aws_cli_error_log() {
  local err_code=$1
  errecho "Error code : $err_code"
```

```
  if [ "$err_code" == 1 ]; then
    errecho "  One or more S3 transfers failed."
  elif [ "$err_code" == 2 ]; then
    errecho "  Command line failed to parse."
  elif [ "$err_code" == 130 ]; then
    errecho "  Process received SIGINT."
  elif [ "$err_code" == 252 ]; then
    errecho "  Command syntax invalid."
  elif [ "$err_code" == 253 ]; then
    errecho "  The system environment or configuration was invalid."
  elif [ "$err_code" == 254 ]; then
    errecho "  The service returned an error."
  elif [ "$err_code" == 255 ]; then
    errecho "  255 is a catch-all error."
  fi

  return 0
}
```

- For API details, see the following topics in *AWS CLI Command Reference*.
  - BatchWriteItem
  - CreateTable
  - DeleteItem
  - DeleteTable
  - DescribeTable
  - GetItem
  - PutItem
  - Query
  - Scan
  - UpdateItem

C++

SDK for C++

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    {
        Aws::Client::ClientConfiguration clientConfig;
        //  1. Create a table with partition: year (N) and sort: title (S).
 (CreateTable)
        if (AwsDoc::DynamoDB::createMoviesDynamoDBTable(clientConfig)) {

            AwsDoc::DynamoDB::dynamodbGettingStartedScenario(clientConfig);

            // 9. Delete the table. (DeleteTable)
            AwsDoc::DynamoDB::deleteMoviesDynamoDBTable(clientConfig);
        }
    }

//! Scenario to modify and query a DynamoDB table.
/*!
  \sa dynamodbGettingStartedScenario()
  \param clientConfiguration: AWS client configuration.
  \return bool: Function succeeded.
 */
bool AwsDoc::DynamoDB::dynamodbGettingStartedScenario(
        const Aws::Client::ClientConfiguration &clientConfiguration) {
    std::cout << std::setfill('*') << std::setw(ASTERISK_FILL_WIDTH) << " "
              << std::endl;
    std::cout << "Welcome to the Amazon DynamoDB getting started demo." <<
 std::endl;
    std::cout << std::setfill('*') << std::setw(ASTERISK_FILL_WIDTH) << " "
              << std::endl;

    Aws::DynamoDB::DynamoDBClient dynamoClient(clientConfiguration);

    // 2. Add a new movie.
    Aws::String title;
```

```
   float rating;
   int year;
   Aws::String plot;
   {
       title = askQuestion(
               "Enter the title of a movie you want to add to the table: ");
       year = askQuestionForInt("What year was it released? ");
       rating = askQuestionForFloatRange("On a scale of 1 - 10, how do you rate
it? ",
                                         1, 10);
       plot = askQuestion("Summarize the plot for me: ");

       Aws::DynamoDB::Model::PutItemRequest putItemRequest;
       putItemRequest.SetTableName(MOVIE_TABLE_NAME);

       putItemRequest.AddItem(YEAR_KEY,

Aws::DynamoDB::Model::AttributeValue().SetN(year));
       putItemRequest.AddItem(TITLE_KEY,

Aws::DynamoDB::Model::AttributeValue().SetS(title));

       // Create attribute for the info map.
       Aws::DynamoDB::Model::AttributeValue infoMapAttribute;

       std::shared_ptr<Aws::DynamoDB::Model::AttributeValue> ratingAttribute =
Aws::MakeShared<Aws::DynamoDB::Model::AttributeValue>(
               ALLOCATION_TAG.c_str());
       ratingAttribute->SetN(rating);
       infoMapAttribute.AddMEntry(RATING_KEY, ratingAttribute);

       std::shared_ptr<Aws::DynamoDB::Model::AttributeValue> plotAttribute =
Aws::MakeShared<Aws::DynamoDB::Model::AttributeValue>(
               ALLOCATION_TAG.c_str());
       plotAttribute->SetS(plot);
       infoMapAttribute.AddMEntry(PLOT_KEY, plotAttribute);

       putItemRequest.AddItem(INFO_KEY, infoMapAttribute);

       Aws::DynamoDB::Model::PutItemOutcome outcome = dynamoClient.PutItem(
               putItemRequest);
       if (!outcome.IsSuccess()) {
           std::cerr << "Failed to add an item: " <<
outcome.GetError().GetMessage()
```

```
                    << std::endl;
            return false;
        }
    }

    std::cout << "\nAdded '" << title << "' to '" << MOVIE_TABLE_NAME << "'."
              << std::endl;

    // 3. Update the rating and plot of the movie by using an update expression.
    {
        rating = askQuestionForFloatRange(
                Aws::String("\nLet's update your movie.\nYou rated it  ") +
                std::to_string(rating)
                + ", what new rating would you give it? ", 1, 10);
        plot = askQuestion(Aws::String("You summarized the plot as '") + plot +
                           "'.\nWhat would you say now? ");

        Aws::DynamoDB::Model::UpdateItemRequest request;
        request.SetTableName(MOVIE_TABLE_NAME);
        request.AddKey(TITLE_KEY,
Aws::DynamoDB::Model::AttributeValue().SetS(title));
        request.AddKey(YEAR_KEY,
Aws::DynamoDB::Model::AttributeValue().SetN(year));
        std::stringstream expressionStream;
        expressionStream << "set " << INFO_KEY << "." << RATING_KEY << " =:r, "
                         << INFO_KEY << "." << PLOT_KEY << " =:p";
        request.SetUpdateExpression(expressionStream.str());
        request.SetExpressionAttributeValues({
                                                     {":r",
Aws::DynamoDB::Model::AttributeValue().SetN(
                                                             rating)},
                                                     {":p",
Aws::DynamoDB::Model::AttributeValue().SetS(
                                                             plot)}
                                             });

        request.SetReturnValues(Aws::DynamoDB::Model::ReturnValue::UPDATED_NEW);

        const Aws::DynamoDB::Model::UpdateItemOutcome &result =
dynamoClient.UpdateItem(
                request);
        if (!result.IsSuccess()) {
            std::cerr << "Error updating movie " + result.GetError().GetMessage()
                      << std::endl;
```

```
            return false;
        }
    }

    std::cout << "\nUpdated '" << title << "' with new attributes:" << std::endl;

    // 4. Put 250 movies in the table from moviedata.json.
    {
        std::cout << "Adding movies from a json file to the database." <<
std::endl;
        const size_t MAX_SIZE_FOR_BATCH_WRITE = 25;
        const size_t MOVIES_TO_WRITE = 10 * MAX_SIZE_FOR_BATCH_WRITE;
        Aws::String jsonString = getMovieJSON();
        if (!jsonString.empty()) {
            Aws::Utils::Json::JsonValue json(jsonString);
            Aws::Utils::Array<Aws::Utils::Json::JsonView> movieJsons =
json.View().AsArray();
            Aws::Vector<Aws::DynamoDB::Model::WriteRequest> writeRequests;

            // To add movies with a cross-section of years, use an appropriate
increment
            // value for iterating through the database.
            size_t increment = movieJsons.GetLength() / MOVIES_TO_WRITE;
            for (size_t i = 0; i < movieJsons.GetLength(); i += increment) {
                writeRequests.push_back(Aws::DynamoDB::Model::WriteRequest());
                Aws::Map<Aws::String, Aws::DynamoDB::Model::AttributeValue>
putItems = movieJsonViewToAttributeMap(
                        movieJsons[i]);
                Aws::DynamoDB::Model::PutRequest putRequest;
                putRequest.SetItem(putItems);
                writeRequests.back().SetPutRequest(putRequest);
                if (writeRequests.size() == MAX_SIZE_FOR_BATCH_WRITE) {
                    Aws::DynamoDB::Model::BatchWriteItemRequest request;
                    request.AddRequestItems(MOVIE_TABLE_NAME, writeRequests);
                    const Aws::DynamoDB::Model::BatchWriteItemOutcome &outcome =
dynamoClient.BatchWriteItem(
                            request);
                    if (!outcome.IsSuccess()) {
                        std::cerr << "Unable to batch write movie data: "
                                  << outcome.GetError().GetMessage()
                                  << std::endl;
                        writeRequests.clear();
                        break;
                    }
```

```
                    else {
                        std::cout << "Added batch of " << writeRequests.size()
                                  << " movies to the database."
                                  << std::endl;
                    }
                    writeRequests.clear();
                }
            }
        }

    std::cout << std::setfill('*') << std::setw(ASTERISK_FILL_WIDTH) << " "
              << std::endl;

    // 5. Get a movie by Key (partition + sort).
    {
        Aws::String titleToGet("King Kong");
        Aws::String answer = askQuestion(Aws::String(
                "Let's move on...Would you like to get info about '" + titleToGet
+
                "'? (y/n) "));
        if (answer == "y") {
            Aws::DynamoDB::Model::GetItemRequest request;
            request.SetTableName(MOVIE_TABLE_NAME);
            request.AddKey(TITLE_KEY,

Aws::DynamoDB::Model::AttributeValue().SetS(titleToGet));
            request.AddKey(YEAR_KEY,
Aws::DynamoDB::Model::AttributeValue().SetN(1933));

            const Aws::DynamoDB::Model::GetItemOutcome &result =
dynamoClient.GetItem(
                    request);
            if (!result.IsSuccess()) {
                std::cerr << "Error " << result.GetError().GetMessage();
            }
            else {
                const Aws::Map<Aws::String, Aws::DynamoDB::Model::AttributeValue>
&item = result.GetResult().GetItem();
                if (!item.empty()) {
                    std::cout << "\nHere's what I found:" << std::endl;
                    printMovieInfo(item);
                }
                else {
```

```
                    std::cout << "\nThe movie was not found in the database."
                              << std::endl;
                }
            }
        }
    }

    // 6. Use Query with a key condition expression to return all movies
    //    released in a given year.
    Aws::String doAgain = "n";
    do {
        Aws::DynamoDB::Model::QueryRequest req;

        req.SetTableName(MOVIE_TABLE_NAME);

        // "year" is a DynamoDB reserved keyword and must be replaced with an
        // expression attribute name.
        req.SetKeyConditionExpression("#dynobase_year = :valueToMatch");
        req.SetExpressionAttributeNames({{"#dynobase_year", YEAR_KEY}});

        int yearToMatch = askQuestionForIntRange(
                "\nLet's get a list of movies released in"
                " a given year. Enter a year between 1972 and 2018 ",
                1972, 2018);
        Aws::Map<Aws::String, Aws::DynamoDB::Model::AttributeValue>
attributeValues;
        attributeValues.emplace(":valueToMatch",
                                Aws::DynamoDB::Model::AttributeValue().SetN(
                                        yearToMatch));
        req.SetExpressionAttributeValues(attributeValues);

        const Aws::DynamoDB::Model::QueryOutcome &result =
dynamoClient.Query(req);
        if (result.IsSuccess()) {
            const Aws::Vector<Aws::Map<Aws::String,
Aws::DynamoDB::Model::AttributeValue>> &items = result.GetResult().GetItems();
            if (!items.empty()) {
                std::cout << "\nThere were " << items.size()
                          << " movies in the database from "
                          << yearToMatch << "." << std::endl;
                for (const auto &item: items) {
                    printMovieInfo(item);
                }
                doAgain = "n";
```

```
            }
            else {
                std::cout << "\nNo movies from " << yearToMatch
                          << " were found in the database"
                          << std::endl;
                doAgain = askQuestion(Aws::String("Try another year? (y/n) "));
            }
        }
        else {
            std::cerr << "Failed to Query items: " <<
result.GetError().GetMessage()
                      << std::endl;
        }

    } while (doAgain == "y");

    //  7. Use Scan to return movies released within a range of years.
    //     Show how to paginate data using ExclusiveStartKey. (Scan +
FilterExpression)
    {
        int startYear = askQuestionForIntRange("\nNow let's scan a range of years
"
                                               "for movies in the database. Enter
a start year: ",
                                               1972, 2018);
        int endYear = askQuestionForIntRange("\nEnter an end year: ",
                                             startYear, 2018);
        Aws::Map<Aws::String, Aws::DynamoDB::Model::AttributeValue>
exclusiveStartKey;
        do {
            Aws::DynamoDB::Model::ScanRequest scanRequest;
            scanRequest.SetTableName(MOVIE_TABLE_NAME);
            scanRequest.SetFilterExpression(
                    "#dynobase_year >= :startYear AND #dynobase_year
<= :endYear");
            scanRequest.SetExpressionAttributeNames({{"#dynobase_year",
YEAR_KEY}});

            Aws::Map<Aws::String, Aws::DynamoDB::Model::AttributeValue>
attributeValues;
            attributeValues.emplace(":startYear",
                                    Aws::DynamoDB::Model::AttributeValue().SetN(
                                            startYear));
            attributeValues.emplace(":endYear",
```

```
                                    Aws::DynamoDB::Model::AttributeValue().SetN(
                                            endYear));
            scanRequest.SetExpressionAttributeValues(attributeValues);

            if (!exclusiveStartKey.empty()) {
                scanRequest.SetExclusiveStartKey(exclusiveStartKey);
            }

            const Aws::DynamoDB::Model::ScanOutcome &result = dynamoClient.Scan(
                    scanRequest);
            if (result.IsSuccess()) {
                const Aws::Vector<Aws::Map<Aws::String,
Aws::DynamoDB::Model::AttributeValue>> &items = result.GetResult().GetItems();
                if (!items.empty()) {
                    std::stringstream stringStream;
                    stringStream << "\nFound " << items.size() << " movies in one
scan."
                                 << " How many would you like to see? ";
                    size_t count = askQuestionForInt(stringStream.str());
                    for (size_t i = 0; i < count && i < items.size(); ++i) {
                        printMovieInfo(items[i]);
                    }
                }
                else {
                    std::cout << "\nNo movies in the database between " <<
startYear <<
                              " and " << endYear << "." << std::endl;
                }

                exclusiveStartKey = result.GetResult().GetLastEvaluatedKey();
                if (!exclusiveStartKey.empty()) {
                    std::cout << "Not all movies were retrieved. Scanning for
more."
                              << std::endl;
                }
                else {
                    std::cout << "All movies were retrieved with this scan."
                              << std::endl;
                }
            }
            else {
                std::cerr << "Failed to Scan movies: "
                          << result.GetError().GetMessage() << std::endl;
            }
```

```
        } while (!exclusiveStartKey.empty());
    }

    // 8. Delete a movie. (DeleteItem)
    {
        std::stringstream stringStream;
        stringStream << "\nWould you like to delete the movie " << title
                     << " from the database? (y/n) ";
        Aws::String answer = askQuestion(stringStream.str());
        if (answer == "y") {
            Aws::DynamoDB::Model::DeleteItemRequest request;
            request.AddKey(YEAR_KEY,
 Aws::DynamoDB::Model::AttributeValue().SetN(year));
            request.AddKey(TITLE_KEY,
                           Aws::DynamoDB::Model::AttributeValue().SetS(title));
            request.SetTableName(MOVIE_TABLE_NAME);

            const Aws::DynamoDB::Model::DeleteItemOutcome &result =
 dynamoClient.DeleteItem(
                    request);
            if (result.IsSuccess()) {
                std::cout << "\nRemoved \"" << title << "\" from the database."
                          << std::endl;
            }
            else {
                std::cerr << "Failed to delete the movie: "
                          << result.GetError().GetMessage()
                          << std::endl;
            }
        }
    }

    return true;
}

//! Routine to convert a JsonView object to an attribute map.
/*!
  \sa movieJsonViewToAttributeMap()
  \param jsonView: Json view object.
  \return map: Map that can be used in a DynamoDB request.
 */
Aws::Map<Aws::String, Aws::DynamoDB::Model::AttributeValue>
AwsDoc::DynamoDB::movieJsonViewToAttributeMap(
        const Aws::Utils::Json::JsonView &jsonView) {
```

```
    Aws::Map<Aws::String, Aws::DynamoDB::Model::AttributeValue> result;

    if (jsonView.KeyExists(YEAR_KEY)) {
        result[YEAR_KEY].SetN(jsonView.GetInteger(YEAR_KEY));
    }
    if (jsonView.KeyExists(TITLE_KEY)) {
        result[TITLE_KEY].SetS(jsonView.GetString(TITLE_KEY));
    }
    if (jsonView.KeyExists(INFO_KEY)) {
        Aws::Map<Aws::String, const
 std::shared_ptr<Aws::DynamoDB::Model::AttributeValue>> infoMap;
        Aws::Utils::Json::JsonView infoView = jsonView.GetObject(INFO_KEY);
        if (infoView.KeyExists(RATING_KEY)) {
            std::shared_ptr<Aws::DynamoDB::Model::AttributeValue> attributeValue
 = std::make_shared<Aws::DynamoDB::Model::AttributeValue>();
            attributeValue->SetN(infoView.GetDouble(RATING_KEY));
            infoMap.emplace(std::make_pair(RATING_KEY, attributeValue));
        }
        if (infoView.KeyExists(PLOT_KEY)) {
            std::shared_ptr<Aws::DynamoDB::Model::AttributeValue> attributeValue
 = std::make_shared<Aws::DynamoDB::Model::AttributeValue>();
            attributeValue->SetS(infoView.GetString(PLOT_KEY));
            infoMap.emplace(std::make_pair(PLOT_KEY, attributeValue));
        }

        result[INFO_KEY].SetM(infoMap);
    }

    return result;
}

//! Create a DynamoDB table to be used in sample code scenarios.
/*!
  \sa createMoviesDynamoDBTable()
  \param clientConfiguration: AWS client configuration.
  \return bool: Function succeeded.
*/
bool AwsDoc::DynamoDB::createMoviesDynamoDBTable(
        const Aws::Client::ClientConfiguration &clientConfiguration) {
    Aws::DynamoDB::DynamoDBClient dynamoClient(clientConfiguration);

    bool movieTableAlreadyExisted = false;

    {
```

```
        Aws::DynamoDB::Model::CreateTableRequest request;

        Aws::DynamoDB::Model::AttributeDefinition yearAttributeDefinition;
        yearAttributeDefinition.SetAttributeName(YEAR_KEY);
        yearAttributeDefinition.SetAttributeType(
                Aws::DynamoDB::Model::ScalarAttributeType::N);
        request.AddAttributeDefinitions(yearAttributeDefinition);

        Aws::DynamoDB::Model::AttributeDefinition titleAttributeDefinition;
        yearAttributeDefinition.SetAttributeName(TITLE_KEY);
        yearAttributeDefinition.SetAttributeType(
                Aws::DynamoDB::Model::ScalarAttributeType::S);
        request.AddAttributeDefinitions(yearAttributeDefinition);

        Aws::DynamoDB::Model::KeySchemaElement yearKeySchema;
        yearKeySchema.WithAttributeName(YEAR_KEY).WithKeyType(
                Aws::DynamoDB::Model::KeyType::HASH);
        request.AddKeySchema(yearKeySchema);

        Aws::DynamoDB::Model::KeySchemaElement titleKeySchema;
        yearKeySchema.WithAttributeName(TITLE_KEY).WithKeyType(
                Aws::DynamoDB::Model::KeyType::RANGE);
        request.AddKeySchema(yearKeySchema);

        Aws::DynamoDB::Model::ProvisionedThroughput throughput;
        throughput.WithReadCapacityUnits(
                PROVISIONED_THROUGHPUT_UNITS).WithWriteCapacityUnits(
                PROVISIONED_THROUGHPUT_UNITS);
        request.SetProvisionedThroughput(throughput);
        request.SetTableName(MOVIE_TABLE_NAME);

        std::cout << "Creating table '" << MOVIE_TABLE_NAME << "'..." <<
std::endl;
        const Aws::DynamoDB::Model::CreateTableOutcome &result =
dynamoClient.CreateTable(
                request);
        if (!result.IsSuccess()) {
            if (result.GetError().GetErrorType() ==
                Aws::DynamoDB::DynamoDBErrors::RESOURCE_IN_USE) {
                std::cout << "Table already exists." << std::endl;
                movieTableAlreadyExisted = true;
            }
            else {
                std::cerr << "Failed to create table: "
```

```
                          << result.GetError().GetMessage();
                return false;
            }
        }
    }

    // Wait for table to become active.
    if (!movieTableAlreadyExisted) {
        std::cout << "Waiting for table '" << MOVIE_TABLE_NAME
                  << "' to become active...." << std::endl;
        if (!AwsDoc::DynamoDB::waitTableActive(MOVIE_TABLE_NAME,
 clientConfiguration)) {
            return false;
        }
        std::cout << "Table '" << MOVIE_TABLE_NAME << "' created and active."
                  << std::endl;
    }

    return true;
}

//! Delete the DynamoDB table used for sample code scenarios.
/*!
  \sa deleteMoviesDynamoDBTable()
  \param clientConfiguration: AWS client configuration.
  \return bool: Function succeeded.
*/
bool AwsDoc::DynamoDB::deleteMoviesDynamoDBTable(
        const Aws::Client::ClientConfiguration &clientConfiguration) {
    Aws::DynamoDB::DynamoDBClient dynamoClient(clientConfiguration);

    Aws::DynamoDB::Model::DeleteTableRequest request;
    request.SetTableName(MOVIE_TABLE_NAME);

    const Aws::DynamoDB::Model::DeleteTableOutcome &result =
 dynamoClient.DeleteTable(
            request);
    if (result.IsSuccess()) {
        std::cout << "Your table \""
                  << result.GetResult().GetTableDescription().GetTableName()
                  << " was deleted.\n";
    }
    else {
        std::cerr << "Failed to delete table: " << result.GetError().GetMessage()
```

```
                  << std::endl;
    }

    return result.IsSuccess();
}

//! Query a newly created DynamoDB table until it is active.
/*!
  \sa waitTableActive()
  \param waitTableActive: The DynamoDB table's name.
  \param dynamoClient: A DynamoDB client.
  \return bool: Function succeeded.
*/
bool AwsDoc::DynamoDB::waitTableActive(const Aws::String &tableName,
                                       const Aws::DynamoDB::DynamoDBClient
 &dynamoClient) {

    // Repeatedly call DescribeTable until table is ACTIVE.
    const int MAX_QUERIES = 20;
    Aws::DynamoDB::Model::DescribeTableRequest request;
    request.SetTableName(tableName);

    int count = 0;
    while (count < MAX_QUERIES) {
        const Aws::DynamoDB::Model::DescribeTableOutcome &result =
 dynamoClient.DescribeTable(
                request);
        if (result.IsSuccess()) {
            Aws::DynamoDB::Model::TableStatus status =
 result.GetResult().GetTable().GetTableStatus();

            if (Aws::DynamoDB::Model::TableStatus::ACTIVE != status) {
                std::this_thread::sleep_for(std::chrono::seconds(1));
            }
            else {
                return true;
            }
        }
        else {
            std::cerr << "Error DynamoDB::waitTableActive "
                      << result.GetError().GetMessage() << std::endl;
            return false;
        }
        count++;
```

```
    }
    return false;
}
```

- For API details, see the following topics in *AWS SDK for C++ API Reference*.
  - BatchWriteItem
  - CreateTable
  - DeleteItem
  - DeleteTable
  - DescribeTable
  - GetItem
  - PutItem
  - Query
  - Scan
  - UpdateItem

Go

SDK for Go V2

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Run an interactive scenario to create the table and perform actions on it.

```
import (
 "context"
 "fmt"
 "log"
 "strings"

 "github.com/aws/aws-sdk-go-v2/aws"
```

```
 "github.com/aws/aws-sdk-go-v2/service/dynamodb"
 "github.com/awsdocs/aws-doc-sdk-examples/gov2/demotools"
 "github.com/awsdocs/aws-doc-sdk-examples/gov2/dynamodb/actions"
)

// RunMovieScenario is an interactive example that shows you how to use the AWS
 SDK for Go
// to create and use an Amazon DynamoDB table that stores data about movies.
//
//  1. Create a table that can hold movie data.
//  2. Put, get, and update a single movie in the table.
//  3. Write movie data to the table from a sample JSON file.
//  4. Query for movies that were released in a given year.
//  5. Scan for movies that were released in a range of years.
//  6. Delete a movie from the table.
//  7. Delete the table.
//
// This example creates a DynamoDB service client from the specified sdkConfig so
 that
// you can replace it with a mocked or stubbed config for unit testing.
//
// It uses a questioner from the `demotools` package to get input during the
 example.
// This package can be found in the ..\..\demotools folder of this repo.
//
// The specified movie sampler is used to get sample data from a URL that is
 loaded
// into the named table.
func RunMovieScenario(
 ctx context.Context, sdkConfig aws.Config, questioner demotools.IQuestioner,
 tableName string,
 movieSampler actions.IMovieSampler) {
 defer func() {
  if r := recover(); r != nil {
   fmt.Printf("Something went wrong with the demo.")
  }
 }()

 log.Println(strings.Repeat("-", 88))
 log.Println("Welcome to the Amazon DynamoDB getting started demo.")
 log.Println(strings.Repeat("-", 88))

 tableBasics := actions.TableBasics{TableName: tableName,
  DynamoDbClient: dynamodb.NewFromConfig(sdkConfig)}
```

```
exists, err := tableBasics.TableExists(ctx)
if err != nil {
 panic(err)
}
if !exists {
 log.Printf("Creating table %v...\n", tableName)
 _, err = tableBasics.CreateMovieTable(ctx)
 if err != nil {
  panic(err)
 } else {
  log.Printf("Created table %v.\n", tableName)
 }
} else {
 log.Printf("Table %v already exists.\n", tableName)
}

var customMovie actions.Movie
customMovie.Title = questioner.Ask("Enter a movie title to add to the table:",
 demotools.NotEmpty{})
customMovie.Year = questioner.AskInt("What year was it released?",
 demotools.NotEmpty{}, demotools.InIntRange{Lower: 1900, Upper: 2030})
customMovie.Info = map[string]interface{}{}
customMovie.Info["rating"] = questioner.AskFloat64(
 "Enter a rating between 1 and 10:",
 demotools.NotEmpty{}, demotools.InFloatRange{Lower: 1, Upper: 10})
customMovie.Info["plot"] = questioner.Ask("What's the plot? ",
 demotools.NotEmpty{})
err = tableBasics.AddMovie(ctx, customMovie)
if err == nil {
 log.Printf("Added %v to the movie table.\n", customMovie.Title)
}
log.Println(strings.Repeat("-", 88))

log.Printf("Let's update your movie. You previously rated it %v.\n",
customMovie.Info["rating"])
customMovie.Info["rating"] = questioner.AskFloat64(
 "What new rating would you give it?",
 demotools.NotEmpty{}, demotools.InFloatRange{Lower: 1, Upper: 10})
log.Printf("You summarized the plot as '%v'.\n", customMovie.Info["plot"])
customMovie.Info["plot"] = questioner.Ask("What would you say now?",
 demotools.NotEmpty{})
attributes, err := tableBasics.UpdateMovie(ctx, customMovie)
if err == nil {
```

```
 log.Printf("Updated %v with new values.\n", customMovie.Title)
 for _, attVal := range attributes {
  for valKey, val := range attVal {
   log.Printf("\t%v: %v\n", valKey, val)
  }
 }
}
log.Println(strings.Repeat("-", 88))

log.Printf("Getting movie data from %v and adding 250 movies to the table...\n",
 movieSampler.GetURL())
movies := movieSampler.GetSampleMovies()
written, err := tableBasics.AddMovieBatch(ctx, movies, 250)
if err != nil {
 panic(err)
} else {
 log.Printf("Added %v movies to the table.\n", written)
}

show := 10
if show > written {
 show = written
}
log.Printf("The first %v movies in the table are:", show)
for index, movie := range movies[:show] {
 log.Printf("\t%v. %v\n", index+1, movie.Title)
}
movieIndex := questioner.AskInt(
 "Enter the number of a movie to get info about it: ",
 demotools.InIntRange{Lower: 1, Upper: show},
)
movie, err := tableBasics.GetMovie(ctx, movies[movieIndex-1].Title,
movies[movieIndex-1].Year)
if err == nil {
 log.Println(movie)
}
log.Println(strings.Repeat("-", 88))

log.Println("Let's get a list of movies released in a given year.")
releaseYear := questioner.AskInt("Enter a year between 1972 and 2018: ",
 demotools.InIntRange{Lower: 1972, Upper: 2018},
)
releases, err := tableBasics.Query(ctx, releaseYear)
if err == nil {
```

```
 if len(releases) == 0 {
  log.Printf("I couldn't find any movies released in %v!\n", releaseYear)
 } else {
  for _, movie = range releases {
   log.Println(movie)
  }
 }
}
log.Println(strings.Repeat("-", 88))

log.Println("Now let's scan for movies released in a range of years.")
startYear := questioner.AskInt("Enter a year: ",
 demotools.InIntRange{Lower: 1972, Upper: 2018})
endYear := questioner.AskInt("Enter another year: ",
 demotools.InIntRange{Lower: 1972, Upper: 2018})
releases, err = tableBasics.Scan(ctx, startYear, endYear)
if err == nil {
 if len(releases) == 0 {
  log.Printf("I couldn't find any movies released between %v and %v!\n",
startYear, endYear)
 } else {
  log.Printf("Found %v movies. In this list, the plot is <nil> because "+
   "we used a projection expression when scanning for items to return only "+
   "the title, year, and rating.\n", len(releases))
  for _, movie = range releases {
   log.Println(movie)
  }
 }
}
log.Println(strings.Repeat("-", 88))

var tables []string
if questioner.AskBool("Do you want to list all of your tables? (y/n) ", "y") {
 tables, err = tableBasics.ListTables(ctx)
 if err == nil {
  log.Printf("Found %v tables:", len(tables))
  for _, table := range tables {
   log.Printf("\t%v", table)
  }
 }
}
log.Println(strings.Repeat("-", 88))

log.Printf("Let's remove your movie '%v'.\n", customMovie.Title)
```

```
 if questioner.AskBool("Do you want to delete it from the table? (y/n) ", "y") {
  err = tableBasics.DeleteMovie(ctx, customMovie)
 }
 if err == nil {
  log.Printf("Deleted %v.\n", customMovie.Title)
 }

 if questioner.AskBool("Delete the table, too? (y/n)", "y") {
  err = tableBasics.DeleteTable(ctx)
 } else {
  log.Println("Don't forget to delete the table when you're done or you might " +
   "incur charges on your account.")
 }
 if err == nil {
  log.Printf("Deleted table %v.\n", tableBasics.TableName)
 }

 log.Println(strings.Repeat("-", 88))
 log.Println("Thanks for watching!")
 log.Println(strings.Repeat("-", 88))
}
```

Define a Movie struct that is used in this example.

```
import (
 "archive/zip"
 "bytes"
 "encoding/json"
 "fmt"
 "io"
 "log"
 "net/http"

 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/attributevalue"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb/types"
)

// Movie encapsulates data about a movie. Title and Year are the composite
 primary key
```

```
// of the movie in Amazon DynamoDB. Title is the sort key, Year is the partition
 key,
// and Info is additional data.
type Movie struct {
 Title string                `dynamodbav:"title"`
 Year  int                   `dynamodbav:"year"`
 Info  map[string]interface{} `dynamodbav:"info"`
}

// GetKey returns the composite primary key of the movie in a format that can be
// sent to DynamoDB.
func (movie Movie) GetKey() map[string]types.AttributeValue {
 title, err := attributevalue.Marshal(movie.Title)
 if err != nil {
  panic(err)
 }
 year, err := attributevalue.Marshal(movie.Year)
 if err != nil {
  panic(err)
 }
 return map[string]types.AttributeValue{"title": title, "year": year}
}

// String returns the title, year, rating, and plot of a movie, formatted for the
 example.
func (movie Movie) String() string {
 return fmt.Sprintf("%v\n\tReleased: %v\n\tRating: %v\n\tPlot: %v\n",
  movie.Title, movie.Year, movie.Info["rating"], movie.Info["plot"])
}
```

Create a struct and methods that call DynamoDB actions.

```
import (
 "context"
 "errors"
 "log"
 "time"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/attributevalue"
```

```
 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/expression"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb/types"
)

// TableBasics encapsulates the Amazon DynamoDB service actions used in the
 examples.
// It contains a DynamoDB service client that is used to act on the specified
 table.
type TableBasics struct {
 DynamoDbClient *dynamodb.Client
 TableName      string
}



// TableExists determines whether a DynamoDB table exists.
func (basics TableBasics) TableExists(ctx context.Context) (bool, error) {
 exists := true
 _, err := basics.DynamoDbClient.DescribeTable(
  ctx, &dynamodb.DescribeTableInput{TableName: aws.String(basics.TableName)},
 )
 if err != nil {
  var notFoundEx *types.ResourceNotFoundException
  if errors.As(err, &notFoundEx) {
   log.Printf("Table %v does not exist.\n", basics.TableName)
   err = nil
  } else {
   log.Printf("Couldn't determine existence of table %v. Here's why: %v\n",
 basics.TableName, err)
  }
  exists = false
 }
 return exists, err
}



// CreateMovieTable creates a DynamoDB table with a composite primary key defined
 as
// a string sort key named `title`, and a numeric partition key named `year`.
// This function uses NewTableExistsWaiter to wait for the table to be created by
// DynamoDB before it returns.
```

```
func (basics TableBasics) CreateMovieTable(ctx context.Context)
 (*types.TableDescription, error) {
 var tableDesc *types.TableDescription
 table, err := basics.DynamoDbClient.CreateTable(ctx, &dynamodb.CreateTableInput{
  AttributeDefinitions: []types.AttributeDefinition{{
   AttributeName: aws.String("year"),
   AttributeType: types.ScalarAttributeTypeN,
  }, {
   AttributeName: aws.String("title"),
   AttributeType: types.ScalarAttributeTypeS,
  }},
  KeySchema: []types.KeySchemaElement{{
   AttributeName: aws.String("year"),
   KeyType:       types.KeyTypeHash,
  }, {
   AttributeName: aws.String("title"),
   KeyType:       types.KeyTypeRange,
  }},
  TableName:   aws.String(basics.TableName),
  BillingMode: types.BillingModePayPerRequest,
 })
 if err != nil {
  log.Printf("Couldn't create table %v. Here's why: %v\n", basics.TableName, err)
 } else {
  waiter := dynamodb.NewTableExistsWaiter(basics.DynamoDbClient)
  err = waiter.Wait(ctx, &dynamodb.DescribeTableInput{
   TableName: aws.String(basics.TableName)}, 5*time.Minute)
  if err != nil {
   log.Printf("Wait for table exists failed. Here's why: %v\n", err)
  }
  tableDesc = table.TableDescription
  log.Printf("Ccreating table test")
 }
 return tableDesc, err
}



// ListTables lists the DynamoDB table names for the current account.
func (basics TableBasics) ListTables(ctx context.Context) ([]string, error) {
 var tableNames []string
 var output *dynamodb.ListTablesOutput
 var err error
```

```
 tablePaginator := dynamodb.NewListTablesPaginator(basics.DynamoDbClient,
 &dynamodb.ListTablesInput{})
 for tablePaginator.HasMorePages() {
  output, err = tablePaginator.NextPage(ctx)
  if err != nil {
   log.Printf("Couldn't list tables. Here's why: %v\n", err)
   break
  } else {
   tableNames = append(tableNames, output.TableNames...)
  }
 }
 return tableNames, err
}



// AddMovie adds a movie the DynamoDB table.
func (basics TableBasics) AddMovie(ctx context.Context, movie Movie) error {
 item, err := attributevalue.MarshalMap(movie)
 if err != nil {
  panic(err)
 }
 _, err = basics.DynamoDbClient.PutItem(ctx, &dynamodb.PutItemInput{
  TableName: aws.String(basics.TableName), Item: item,
 })
 if err != nil {
  log.Printf("Couldn't add item to table. Here's why: %v\n", err)
 }
 return err
}



// UpdateMovie updates the rating and plot of a movie that already exists in the
// DynamoDB table. This function uses the `expression` package to build the
 update
// expression.
func (basics TableBasics) UpdateMovie(ctx context.Context, movie Movie)
 (map[string]map[string]interface{}, error) {
 var err error
 var response *dynamodb.UpdateItemOutput
 var attributeMap map[string]map[string]interface{}
 update := expression.Set(expression.Name("info.rating"),
 expression.Value(movie.Info["rating"]))
```

```
 update.Set(expression.Name("info.plot"), expression.Value(movie.Info["plot"]))
 expr, err := expression.NewBuilder().WithUpdate(update).Build()
 if err != nil {
  log.Printf("Couldn't build expression for update. Here's why: %v\n", err)
 } else {
  response, err = basics.DynamoDbClient.UpdateItem(ctx,
 &dynamodb.UpdateItemInput{
   TableName:                 aws.String(basics.TableName),
   Key:                       movie.GetKey(),
   ExpressionAttributeNames:  expr.Names(),
   ExpressionAttributeValues: expr.Values(),
   UpdateExpression:          expr.Update(),
   ReturnValues:              types.ReturnValueUpdatedNew,
  })
  if err != nil {
   log.Printf("Couldn't update movie %v. Here's why: %v\n", movie.Title, err)
  } else {
   err = attributevalue.UnmarshalMap(response.Attributes, &attributeMap)
   if err != nil {
    log.Printf("Couldn't unmarshall update response. Here's why: %v\n", err)
   }
  }
 }
 return attributeMap, err
}



// AddMovieBatch adds a slice of movies to the DynamoDB table. The function sends
// batches of 25 movies to DynamoDB until all movies are added or it reaches the
// specified maximum.
func (basics TableBasics) AddMovieBatch(ctx context.Context, movies []Movie,
 maxMovies int) (int, error) {
 var err error
 var item map[string]types.AttributeValue
 written := 0
 batchSize := 25 // DynamoDB allows a maximum batch size of 25 items.
 start := 0
 end := start + batchSize
 for start < maxMovies && start < len(movies) {
  var writeReqs []types.WriteRequest
  if end > len(movies) {
   end = len(movies)
  }
```

```
  for _, movie := range movies[start:end] {
   item, err = attributevalue.MarshalMap(movie)
   if err != nil {
    log.Printf("Couldn't marshal movie %v for batch writing. Here's why: %v\n",
 movie.Title, err)
   } else {
    writeReqs = append(
     writeReqs,
     types.WriteRequest{PutRequest: &types.PutRequest{Item: item}},
    )
   }
  }
  _, err = basics.DynamoDbClient.BatchWriteItem(ctx,
 &dynamodb.BatchWriteItemInput{
   RequestItems: map[string][]types.WriteRequest{basics.TableName: writeReqs}})
  if err != nil {
   log.Printf("Couldn't add a batch of movies to %v. Here's why: %v\n",
 basics.TableName, err)
  } else {
   written += len(writeReqs)
  }
  start = end
  end += batchSize
 }

 return written, err
}



// GetMovie gets movie data from the DynamoDB table by using the primary
 composite key
// made of title and year.
func (basics TableBasics) GetMovie(ctx context.Context, title string, year int)
 (Movie, error) {
 movie := Movie{Title: title, Year: year}
 response, err := basics.DynamoDbClient.GetItem(ctx, &dynamodb.GetItemInput{
  Key: movie.GetKey(), TableName: aws.String(basics.TableName),
 })
 if err != nil {
  log.Printf("Couldn't get info about %v. Here's why: %v\n", title, err)
 } else {
  err = attributevalue.UnmarshalMap(response.Item, &movie)
  if err != nil {
```

```
   log.Printf("Couldn't unmarshal response. Here's why: %v\n", err)
  }
 }
 return movie, err
}



// Query gets all movies in the DynamoDB table that were released in the
 specified year.
// The function uses the `expression` package to build the key condition
 expression
// that is used in the query.
func (basics TableBasics) Query(ctx context.Context, releaseYear int) ([]Movie,
 error) {
 var err error
 var response *dynamodb.QueryOutput
 var movies []Movie
 keyEx := expression.Key("year").Equal(expression.Value(releaseYear))
 expr, err := expression.NewBuilder().WithKeyCondition(keyEx).Build()
 if err != nil {
  log.Printf("Couldn't build expression for query. Here's why: %v\n", err)
 } else {
  queryPaginator := dynamodb.NewQueryPaginator(basics.DynamoDbClient,
 &dynamodb.QueryInput{
   TableName:                 aws.String(basics.TableName),
   ExpressionAttributeNames:  expr.Names(),
   ExpressionAttributeValues: expr.Values(),
   KeyConditionExpression:    expr.KeyCondition(),
  })
  for queryPaginator.HasMorePages() {
   response, err = queryPaginator.NextPage(ctx)
   if err != nil {
    log.Printf("Couldn't query for movies released in %v. Here's why: %v\n",
 releaseYear, err)
    break
   } else {
    var moviePage []Movie
    err = attributevalue.UnmarshalListOfMaps(response.Items, &moviePage)
    if err != nil {
     log.Printf("Couldn't unmarshal query response. Here's why: %v\n", err)
     break
    } else {
     movies = append(movies, moviePage...)
```

```
    }
   }
  }
 }
 return movies, err
}



// Scan gets all movies in the DynamoDB table that were released in a range of
 years
// and projects them to return a reduced set of fields.
// The function uses the `expression` package to build the filter and projection
// expressions.
func (basics TableBasics) Scan(ctx context.Context, startYear int, endYear int)
 ([]Movie, error) {
 var movies []Movie
 var err error
 var response *dynamodb.ScanOutput
 filtEx := expression.Name("year").Between(expression.Value(startYear),
 expression.Value(endYear))
 projEx := expression.NamesList(
  expression.Name("year"), expression.Name("title"),
 expression.Name("info.rating"))
 expr, err :=
 expression.NewBuilder().WithFilter(filtEx).WithProjection(projEx).Build()
 if err != nil {
  log.Printf("Couldn't build expressions for scan. Here's why: %v\n", err)
 } else {
  scanPaginator := dynamodb.NewScanPaginator(basics.DynamoDbClient,
 &dynamodb.ScanInput{
   TableName:                 aws.String(basics.TableName),
   ExpressionAttributeNames:  expr.Names(),
   ExpressionAttributeValues: expr.Values(),
   FilterExpression:          expr.Filter(),
   ProjectionExpression:      expr.Projection(),
  })
  for scanPaginator.HasMorePages() {
   response, err = scanPaginator.NextPage(ctx)
   if err != nil {
    log.Printf("Couldn't scan for movies released between %v and %v. Here's why:
 %v\n",
     startYear, endYear, err)
    break
```

```
   } else {
    var moviePage []Movie
    err = attributevalue.UnmarshalListOfMaps(response.Items, &moviePage)
    if err != nil {
     log.Printf("Couldn't unmarshal query response. Here's why: %v\n", err)
     break
    } else {
     movies = append(movies, moviePage...)
    }
   }
  }
 }
 return movies, err
}



// DeleteMovie removes a movie from the DynamoDB table.
func (basics TableBasics) DeleteMovie(ctx context.Context, movie Movie) error {
 _, err := basics.DynamoDbClient.DeleteItem(ctx, &dynamodb.DeleteItemInput{
  TableName: aws.String(basics.TableName), Key: movie.GetKey(),
 })
 if err != nil {
  log.Printf("Couldn't delete %v from the table. Here's why: %v\n", movie.Title,
 err)
 }
 return err
}



// DeleteTable deletes the DynamoDB table and all of its data.
func (basics TableBasics) DeleteTable(ctx context.Context) error {
 _, err := basics.DynamoDbClient.DeleteTable(ctx, &dynamodb.DeleteTableInput{
  TableName: aws.String(basics.TableName)})
 if err != nil {
  log.Printf("Couldn't delete table %v. Here's why: %v\n", basics.TableName, err)
 }
 return err
}
```

- For API details, see the following topics in *AWS SDK for Go API Reference*.

- BatchWriteItem
- CreateTable
- DeleteItem
- DeleteTable
- DescribeTable
- GetItem
- PutItem
- Query
- Scan
- UpdateItem

Java

SDK for Java 2.x

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Create a DynamoDB table.

```
// Create a table with a Sort key.
public static void createTable(DynamoDbClient ddb, String tableName) {
    DynamoDbWaiter dbWaiter = ddb.waiter();
    ArrayList<AttributeDefinition> attributeDefinitions = new ArrayList<>();

    // Define attributes.
    attributeDefinitions.add(AttributeDefinition.builder()
        .attributeName("year")
        .attributeType("N")
        .build());

    attributeDefinitions.add(AttributeDefinition.builder()
        .attributeName("title")
        .attributeType("S")
        .build());
```

```
        ArrayList<KeySchemaElement> tableKey = new ArrayList<>();
        KeySchemaElement key = KeySchemaElement.builder()
            .attributeName("year")
            .keyType(KeyType.HASH)
            .build();

        KeySchemaElement key2 = KeySchemaElement.builder()
            .attributeName("title")
            .keyType(KeyType.RANGE)
            .build();

        // Add KeySchemaElement objects to the list.
        tableKey.add(key);
        tableKey.add(key2);

        CreateTableRequest request = CreateTableRequest.builder()
            .keySchema(tableKey)
            .billingMode(BillingMode.PAY_PER_REQUEST) //  DynamoDB automatically
scales based on traffic.
            .attributeDefinitions(attributeDefinitions)
            .tableName(tableName)
            .build();

        try {
            CreateTableResponse response = ddb.createTable(request);
            DescribeTableRequest tableRequest = DescribeTableRequest.builder()
                .tableName(tableName)
                .build();

            // Wait until the Amazon DynamoDB table is created.
            WaiterResponse<DescribeTableResponse> waiterResponse =
dbWaiter.waitUntilTableExists(tableRequest);
            waiterResponse.matched().response().ifPresent(System.out::println);
            String newTable = response.tableDescription().tableName();
            System.out.println("The " + newTable + " was successfully created.");

        } catch (DynamoDbException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }
```

Create a helper function to download and extract the sample JSON file.

```
   // Load data into the table.
   public static void loadData(DynamoDbClient ddb, String tableName, String
fileName) throws IOException {
       DynamoDbEnhancedClient enhancedClient = DynamoDbEnhancedClient.builder()
           .dynamoDbClient(ddb)
           .build();

       DynamoDbTable<Movies> mappedTable = enhancedClient.table("Movies",
TableSchema.fromBean(Movies.class));
       JsonParser parser = new JsonFactory().createParser(new File(fileName));
       com.fasterxml.jackson.databind.JsonNode rootNode = new
ObjectMapper().readTree(parser);
       Iterator<JsonNode> iter = rootNode.iterator();
       ObjectNode currentNode;
       int t = 0;
       while (iter.hasNext()) {
           // Only add 200 Movies to the table.
           if (t == 200)
               break;
           currentNode = (ObjectNode) iter.next();

           int year = currentNode.path("year").asInt();
           String title = currentNode.path("title").asText();
           String info = currentNode.path("info").toString();

           Movies movies = new Movies();
           movies.setYear(year);
           movies.setTitle(title);
           movies.setInfo(info);

           // Put the data into the Amazon DynamoDB Movie table.
           mappedTable.putItem(movies);
           t++;
       }
   }
```

Get an item from a table.

```
   public static void getItem(DynamoDbClient ddb) {
```

```
        HashMap<String, AttributeValue> keyToGet = new HashMap<>();
        keyToGet.put("year", AttributeValue.builder()
            .n("1933")
            .build());

        keyToGet.put("title", AttributeValue.builder()
            .s("King Kong")
            .build());

        GetItemRequest request = GetItemRequest.builder()
            .key(keyToGet)
            .tableName("Movies")
            .build();

        try {
            Map<String, AttributeValue> returnedItem =
ddb.getItem(request).item();

            if (returnedItem != null) {
                Set<String> keys = returnedItem.keySet();
                System.out.println("Amazon DynamoDB table attributes: \n");

                for (String key1 : keys) {
                    System.out.format("%s: %s\n", key1,
returnedItem.get(key1).toString());
                }
            } else {
                System.out.format("No item found with the key %s!\n", "year");
            }

        } catch (DynamoDbException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }
```

Full example.

```
/**
 * Before running this Java V2 code example, set up your development
 * environment, including your credentials.
```

```
 * <p>
 * For more information, see the following documentation topic:
 * <p>
 * https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-
started.html
 * <p>
 * This Java example performs these tasks:
 * <p>
 * 1. Creates the Amazon DynamoDB Movie table with partition and sort key.
 * 2. Puts data into the Amazon DynamoDB table from a JSON document using the
 * Enhanced client.
 * 3. Gets data from the Movie table.
 * 4. Adds a new item.
 * 5. Updates an item.
 * 6. Uses a Scan to query items using the Enhanced client.
 * 7. Queries all items where the year is 2013 using the Enhanced Client.
 * 8. Deletes the table.
 */

public class Scenario {
    public static final String DASHES = new String(new char[80]).replace("\0",
 "-");

    public static void main(String[] args) throws IOException {
        String tableName = "Movies";
        String fileName = "../../../resources/sample_files/movies.json";
        Region region = Region.US_EAST_1;
        DynamoDbClient ddb = DynamoDbClient.builder()
            .region(region)
            .build();

        System.out.println(DASHES);
        System.out.println("Welcome to the Amazon DynamoDB example scenario.");
        System.out.println(DASHES);

        System.out.println(DASHES);
        System.out.println(
            "1. Creating an Amazon DynamoDB table named Movies with a key named
 year and a sort key named title.");
        createTable(ddb, tableName);
        System.out.println(DASHES);

        System.out.println(DASHES);
        System.out.println("2. Loading data into the Amazon DynamoDB table.");
```

```
        loadData(ddb, tableName, fileName);
        System.out.println(DASHES);

        System.out.println(DASHES);
        System.out.println("3. Getting data from the Movie table.");
        getItem(ddb);
        System.out.println(DASHES);

        System.out.println(DASHES);
        System.out.println("4. Putting a record into the Amazon DynamoDB
table.");
        putRecord(ddb);
        System.out.println(DASHES);

        System.out.println(DASHES);
        System.out.println("5. Updating a record.");
        updateTableItem(ddb, tableName);
        System.out.println(DASHES);

        System.out.println(DASHES);
        System.out.println("6. Scanning the Amazon DynamoDB table.");
        scanMovies(ddb, tableName);
        System.out.println(DASHES);

        System.out.println(DASHES);
        System.out.println("7. Querying the Movies released in 2013.");
        queryTable(ddb);
        System.out.println(DASHES);

        System.out.println(DASHES);
        System.out.println("8. Deleting the Amazon DynamoDB table.");
        deleteDynamoDBTable(ddb, tableName);
        System.out.println(DASHES);

        ddb.close();
    }

    // Create a table with a Sort key.
    public static void createTable(DynamoDbClient ddb, String tableName) {
        DynamoDbWaiter dbWaiter = ddb.waiter();
        ArrayList<AttributeDefinition> attributeDefinitions = new ArrayList<>();

        // Define attributes.
        attributeDefinitions.add(AttributeDefinition.builder()
```

```
            .attributeName("year")
            .attributeType("N")
            .build());

        attributeDefinitions.add(AttributeDefinition.builder()
            .attributeName("title")
            .attributeType("S")
            .build());

        ArrayList<KeySchemaElement> tableKey = new ArrayList<>();
        KeySchemaElement key = KeySchemaElement.builder()
            .attributeName("year")
            .keyType(KeyType.HASH)
            .build();

        KeySchemaElement key2 = KeySchemaElement.builder()
            .attributeName("title")
            .keyType(KeyType.RANGE)
            .build();

        // Add KeySchemaElement objects to the list.
        tableKey.add(key);
        tableKey.add(key2);

        CreateTableRequest request = CreateTableRequest.builder()
            .keySchema(tableKey)
            .billingMode(BillingMode.PAY_PER_REQUEST) //  DynamoDB automatically
scales based on traffic.
            .attributeDefinitions(attributeDefinitions)
            .tableName(tableName)
            .build();

        try {
            CreateTableResponse response = ddb.createTable(request);
            DescribeTableRequest tableRequest = DescribeTableRequest.builder()
                .tableName(tableName)
                .build();

            // Wait until the Amazon DynamoDB table is created.
            WaiterResponse<DescribeTableResponse> waiterResponse =
dbWaiter.waitUntilTableExists(tableRequest);
            waiterResponse.matched().response().ifPresent(System.out::println);
            String newTable = response.tableDescription().tableName();
            System.out.println("The " + newTable + " was successfully created.");
```

```
        } catch (DynamoDbException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }

    // Query the table.
    public static void queryTable(DynamoDbClient ddb) {
        try {
            DynamoDbEnhancedClient enhancedClient =
DynamoDbEnhancedClient.builder()
                .dynamoDbClient(ddb)
                .build();

            DynamoDbTable<Movies> custTable = enhancedClient.table("Movies",
TableSchema.fromBean(Movies.class));
            QueryConditional queryConditional = QueryConditional
                .keyEqualTo(Key.builder()
                    .partitionValue(2013)
                    .build());

            // Get items in the table and write out the ID value.
            Iterator<Movies> results =
custTable.query(queryConditional).items().iterator();
            String result = "";

            while (results.hasNext()) {
                Movies rec = results.next();
                System.out.println("The title of the movie is " +
rec.getTitle());
                System.out.println("The movie information  is " + rec.getInfo());
            }

        } catch (DynamoDbException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }

    // Scan the table.
    public static void scanMovies(DynamoDbClient ddb, String tableName) {
        System.out.println("******* Scanning all movies.\n");
        try {
```

```
            DynamoDbEnhancedClient enhancedClient =
DynamoDbEnhancedClient.builder()
                .dynamoDbClient(ddb)
                .build();

            DynamoDbTable<Movies> custTable = enhancedClient.table("Movies",
TableSchema.fromBean(Movies.class));
            Iterator<Movies> results = custTable.scan().items().iterator();
            while (results.hasNext()) {
                Movies rec = results.next();
                System.out.println("The movie title is " + rec.getTitle());
                System.out.println("The movie year is " + rec.getYear());
            }

        } catch (DynamoDbException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }

    // Load data into the table.
    public static void loadData(DynamoDbClient ddb, String tableName, String
fileName) throws IOException {
        DynamoDbEnhancedClient enhancedClient = DynamoDbEnhancedClient.builder()
            .dynamoDbClient(ddb)
            .build();

        DynamoDbTable<Movies> mappedTable = enhancedClient.table("Movies",
TableSchema.fromBean(Movies.class));
        JsonParser parser = new JsonFactory().createParser(new File(fileName));
        com.fasterxml.jackson.databind.JsonNode rootNode = new
ObjectMapper().readTree(parser);
        Iterator<JsonNode> iter = rootNode.iterator();
        ObjectNode currentNode;
        int t = 0;
        while (iter.hasNext()) {
            // Only add 200 Movies to the table.
            if (t == 200)
                break;
            currentNode = (ObjectNode) iter.next();

            int year = currentNode.path("year").asInt();
            String title = currentNode.path("title").asText();
            String info = currentNode.path("info").toString();
```

```
            Movies movies = new Movies();
            movies.setYear(year);
            movies.setTitle(title);
            movies.setInfo(info);

            // Put the data into the Amazon DynamoDB Movie table.
            mappedTable.putItem(movies);
            t++;
        }
    }

    // Update the record to include show only directors.
    public static void updateTableItem(DynamoDbClient ddb, String tableName) {
        HashMap<String, AttributeValue> itemKey = new HashMap<>();
        itemKey.put("year", AttributeValue.builder().n("1933").build());
        itemKey.put("title", AttributeValue.builder().s("King Kong").build());

        HashMap<String, AttributeValueUpdate> updatedValues = new HashMap<>();
        updatedValues.put("info", AttributeValueUpdate.builder()
            .value(AttributeValue.builder().s("{\"directors\":[\"Merian C. Cooper
\",\"Ernest B. Schoedsack\"]")
                .build())
            .action(AttributeAction.PUT)
            .build());

        UpdateItemRequest request = UpdateItemRequest.builder()
            .tableName(tableName)
            .key(itemKey)
            .attributeUpdates(updatedValues)
            .build();

        try {
            ddb.updateItem(request);
        } catch (ResourceNotFoundException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        } catch (DynamoDbException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }

        System.out.println("Item was updated!");
    }
```

```
   public static void deleteDynamoDBTable(DynamoDbClient ddb, String tableName)
{
       DeleteTableRequest request = DeleteTableRequest.builder()
           .tableName(tableName)
           .build();

       try {
           ddb.deleteTable(request);

       } catch (DynamoDbException e) {
           System.err.println(e.getMessage());
           System.exit(1);
       }
       System.out.println(tableName + " was successfully deleted!");
   }

   public static void putRecord(DynamoDbClient ddb) {
       try {
           DynamoDbEnhancedClient enhancedClient =
DynamoDbEnhancedClient.builder()
               .dynamoDbClient(ddb)
               .build();

           DynamoDbTable<Movies> table = enhancedClient.table("Movies",
TableSchema.fromBean(Movies.class));

           // Populate the Table.
           Movies record = new Movies();
           record.setYear(2020);
           record.setTitle("My Movie2");
           record.setInfo("no info");
           table.putItem(record);

       } catch (DynamoDbException e) {
           System.err.println(e.getMessage());
           System.exit(1);
       }
       System.out.println("Added a new movie to the table.");
   }

   public static void getItem(DynamoDbClient ddb) {

       HashMap<String, AttributeValue> keyToGet = new HashMap<>();
```

```
        keyToGet.put("year", AttributeValue.builder()
            .n("1933")
            .build());

        keyToGet.put("title", AttributeValue.builder()
            .s("King Kong")
            .build());

        GetItemRequest request = GetItemRequest.builder()
            .key(keyToGet)
            .tableName("Movies")
            .build();

        try {
            Map<String, AttributeValue> returnedItem =
 ddb.getItem(request).item();

            if (returnedItem != null) {
                Set<String> keys = returnedItem.keySet();
                System.out.println("Amazon DynamoDB table attributes: \n");

                for (String key1 : keys) {
                    System.out.format("%s: %s\n", key1,
 returnedItem.get(key1).toString());
                }
            } else {
                System.out.format("No item found with the key %s!\n", "year");
            }

        } catch (DynamoDbException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }
}
```

- For API details, see the following topics in *AWS SDK for Java 2.x API Reference*.
  - BatchWriteItem
  - CreateTable
  - DeleteItem
  - DeleteTable

- DescribeTable
- GetItem
- PutItem
- Query
- Scan
- UpdateItem

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import { readFileSync } from "node:fs";
import {
  BillingMode,
  CreateTableCommand,
  DeleteTableCommand,
  DynamoDBClient,
  waitUntilTableExists,
} from "@aws-sdk/client-dynamodb";

/**
 * This module is a convenience library. It abstracts Amazon DynamoDB's data type
 * descriptors (such as S, N, B, and BOOL) by marshalling JavaScript objects into
 * AttributeValue shapes.
 */
import {
  BatchWriteCommand,
  DeleteCommand,
  DynamoDBDocumentClient,
  GetCommand,
  PutCommand,
  UpdateCommand,
  paginateQuery,
```

```
  paginateScan,
} from "@aws-sdk/lib-dynamodb";

// These modules are local to our GitHub repository. We recommend cloning
// the project from GitHub if you want to run this example.
// For more information, see https://github.com/awsdocs/aws-doc-sdk-examples.
import { getUniqueName } from "@aws-doc-sdk-examples/lib/utils/util-string.js";
import { dirnameFromMetaUrl } from "@aws-doc-sdk-examples/lib/utils/util-fs.js";
import { chunkArray } from "@aws-doc-sdk-examples/lib/utils/util-array.js";

const dirname = dirnameFromMetaUrl(import.meta.url);
const tableName = getUniqueName("Movies");
const client = new DynamoDBClient({});
const docClient = DynamoDBDocumentClient.from(client);

const log = (msg) => console.log(`[SCENARIO] ${msg}`);

export const main = async () => {
  /**
   * Create a table.
   */

  const createTableCommand = new CreateTableCommand({
    TableName: tableName,
    // This example performs a large write to the database.
    // Set the billing mode to PAY_PER_REQUEST to
    // avoid throttling the large write.
    BillingMode: BillingMode.PAY_PER_REQUEST,
    // Define the attributes that are necessary for the key schema.
    AttributeDefinitions: [
      {
        AttributeName: "year",
        // 'N' is a data type descriptor that represents a number type.
        // For a list of all data type descriptors, see the following link.
        // https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/
Programming.LowLevelAPI.html#Programming.LowLevelAPI.DataTypeDescriptors
        AttributeType: "N",
      },
      { AttributeName: "title", AttributeType: "S" },
    ],
    // The KeySchema defines the primary key. The primary key can be
    // a partition key, or a combination of a partition key and a sort key.
    // Key schema design is important. For more info, see
```

```
    // https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/best-
practices.html
    KeySchema: [
      // The way your data is accessed determines how you structure your keys.
      // The movies table will be queried for movies by year. It makes sense
      // to make year our partition (HASH) key.
      { AttributeName: "year", KeyType: "HASH" },
      { AttributeName: "title", KeyType: "RANGE" },
    ],
  });

  log("Creating a table.");
  const createTableResponse = await client.send(createTableCommand);
  log(`Table created: ${JSON.stringify(createTableResponse.TableDescription)}`);

  // This polls with DescribeTableCommand until the requested table is 'ACTIVE'.
  // You can't write to a table before it's active.
  log("Waiting for the table to be active.");
  await waitUntilTableExists({ client }, { TableName: tableName });
  log("Table active.");

  /**
   * Add a movie to the table.
   */

  log("Adding a single movie to the table.");
  // PutCommand is the first example usage of 'lib-dynamodb'.
  const putCommand = new PutCommand({
    TableName: tableName,
    Item: {
      // In 'client-dynamodb', the AttributeValue would be required (`year: { N:
 1981 }`)
      // 'lib-dynamodb' simplifies the usage ( `year: 1981` )
      year: 1981,
      // The preceding KeySchema defines 'title' as our sort (RANGE) key, so
 'title'
      // is required.
      title: "The Evil Dead",
      // Every other attribute is optional.
      info: {
        genres: ["Horror"],
      },
    },
  });
```

```
  await docClient.send(putCommand);
  log("The movie was added.");

  /**
   * Get a movie from the table.
   */

  log("Getting a single movie from the table.");
  const getCommand = new GetCommand({
    TableName: tableName,
    // Requires the complete primary key. For the movies table, the primary key
    // is only the id (partition key).
    Key: {
      year: 1981,
      title: "The Evil Dead",
    },
    // Set this to make sure that recent writes are reflected.
    // For more information, see https://docs.aws.amazon.com/amazondynamodb/
latest/developerguide/HowItWorks.ReadConsistency.html.
    ConsistentRead: true,
  });
  const getResponse = await docClient.send(getCommand);
  log(`Got the movie: ${JSON.stringify(getResponse.Item)}`);

  /**
   * Update a movie in the table.
   */

  log("Updating a single movie in the table.");
  const updateCommand = new UpdateCommand({
    TableName: tableName,
    Key: { year: 1981, title: "The Evil Dead" },
    // This update expression appends "Comedy" to the list of genres.
    // For more information on update expressions, see
    // https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/
Expressions.UpdateExpressions.html
    UpdateExpression: "set #i.#g = list_append(#i.#g, :vals)",
    ExpressionAttributeNames: { "#i": "info", "#g": "genres" },
    ExpressionAttributeValues: {
      ":vals": ["Comedy"],
    },
    ReturnValues: "ALL_NEW",
  });
  const updateResponse = await docClient.send(updateCommand);
```

```
log(`Movie updated: ${JSON.stringify(updateResponse.Attributes)}`);

/**
 * Delete a movie from the table.
 */

log("Deleting a single movie from the table.");
const deleteCommand = new DeleteCommand({
  TableName: tableName,
  Key: { year: 1981, title: "The Evil Dead" },
});
await docClient.send(deleteCommand);
log("Movie deleted.");

/**
 * Upload a batch of movies.
 */

log("Adding movies from local JSON file.");
const file = readFileSync(
  `${dirname}../../../../resources/sample_files/movies.json`,
);
const movies = JSON.parse(file.toString());
// chunkArray is a local convenience function. It takes an array and returns
// a generator function. The generator function yields every N items.
const movieChunks = chunkArray(movies, 25);
// For every chunk of 25 movies, make one BatchWrite request.
for (const chunk of movieChunks) {
  const putRequests = chunk.map((movie) => ({
    PutRequest: {
      Item: movie,
    },
  }));

  const command = new BatchWriteCommand({
    RequestItems: {
      [tableName]: putRequests,
    },
  });

  await docClient.send(command);
}
log("Movies added.");
```

```
  /**
   * Query for movies by year.
   */

  log("Querying for all movies from 1981.");
  const paginatedQuery = paginateQuery(
    { client: docClient },
    {
      TableName: tableName,
      //For more information about query expressions, see
      // https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/
Query.html#Query.KeyConditionExpressions
      KeyConditionExpression: "#y = :y",
      // 'year' is a reserved word in DynamoDB. Indicate that it's an attribute
      // name by using an expression attribute name.
      ExpressionAttributeNames: { "#y": "year" },
      ExpressionAttributeValues: { ":y": 1981 },
      ConsistentRead: true,
    },
  );
  /**
   * @type { Record<string, any>[] };
   */
  const movies1981 = [];
  for await (const page of paginatedQuery) {
    movies1981.push(...page.Items);
  }
  log(`Movies: ${movies1981.map((m) => m.title).join(", ")}`);

  /**
   * Scan the table for movies between 1980 and 1990.
   */

  log("Scan for movies released between 1980 and 1990");
  // A 'Scan' operation always reads every item in the table. If your design
 requires
  // the use of 'Scan', consider indexing your table or changing your design.
  // https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/bp-query-
scan.html
  const paginatedScan = paginateScan(
    { client: docClient },
    {
      TableName: tableName,
```

```
      // Scan uses a filter expression instead of a key condition expression.
 Scan will
      // read the entire table and then apply the filter.
      FilterExpression: "#y between :y1 and :y2",
      ExpressionAttributeNames: { "#y": "year" },
      ExpressionAttributeValues: { ":y1": 1980, ":y2": 1990 },
      ConsistentRead: true,
    },
  );
  /**
   * @type { Record<string, any>[] };
   */
  const movies1980to1990 = [];
  for await (const page of paginatedScan) {
    movies1980to1990.push(...page.Items);
  }
  log(
    `Movies: ${movies1980to1990
      .map((m) => `${m.title} (${m.year})`)
      .join(", ")}`,
  );

  /**
   * Delete the table.
   */

  const deleteTableCommand = new DeleteTableCommand({ TableName: tableName });
  log(`Deleting table ${tableName}.`);
  await client.send(deleteTableCommand);
  log("Table deleted.");
};
```

- For API details, see the following topics in *AWS SDK for JavaScript API Reference*.
  - BatchWriteItem
  - CreateTable
  - DeleteItem
  - DeleteTable
  - DescribeTable
  - GetItem

- PutItem
- Query
- Scan
- UpdateItem

Kotlin

SDK for Kotlin

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Create a DynamoDB table.

```
suspend fun createScenarioTable(
    tableNameVal: String,
    key: String,
) {
    val attDef =
        AttributeDefinition {
            attributeName = key
            attributeType = ScalarAttributeType.N
        }

    val attDef1 =
        AttributeDefinition {
            attributeName = "title"
            attributeType = ScalarAttributeType.S
        }

    val keySchemaVal =
        KeySchemaElement {
            attributeName = key
            keyType = KeyType.Hash
        }

    val keySchemaVal1 =
```

```
        KeySchemaElement {
            attributeName = "title"
            keyType = KeyType.Range
        }

    val request =
        CreateTableRequest {
            attributeDefinitions = listOf(attDef, attDef1)
            keySchema = listOf(keySchemaVal, keySchemaVal1)
            billingMode = BillingMode.PayPerRequest
            tableName = tableNameVal
        }

    DynamoDbClient.fromEnvironment { region = "us-east-1" }.use { ddb ->
        val response = ddb.createTable(request)
        ddb.waitUntilTableExists {
            // suspend call
            tableName = tableNameVal
        }
        println("The table was successfully created
 ${response.tableDescription?.tableArn}")
    }
}
```

Create a helper function to download and extract the sample JSON file.

```
// Load data into the table.
suspend fun loadData(
    tableName: String,
    fileName: String,
) {
    val parser = JsonFactory().createParser(File(fileName))
    val rootNode = ObjectMapper().readTree<JsonNode>(parser)
    val iter: Iterator<JsonNode> = rootNode.iterator()
    var currentNode: ObjectNode

    var t = 0
    while (iter.hasNext()) {
        if (t == 50) {
            break
        }
```

```
        currentNode = iter.next() as ObjectNode
        val year = currentNode.path("year").asInt()
        val title = currentNode.path("title").asText()
        val info = currentNode.path("info").toString()
        putMovie(tableName, year, title, info)
        t++
    }
}

suspend fun putMovie(
    tableNameVal: String,
    year: Int,
    title: String,
    info: String,
) {
    val itemValues = mutableMapOf<String, AttributeValue>()
    val strVal = year.toString()
    // Add all content to the table.
    itemValues["year"] = AttributeValue.N(strVal)
    itemValues["title"] = AttributeValue.S(title)
    itemValues["info"] = AttributeValue.S(info)

    val request =
        PutItemRequest {
            tableName = tableNameVal
            item = itemValues
        }

    DynamoDbClient.fromEnvironment { region = "us-east-1" }.use { ddb ->
        ddb.putItem(request)
        println("Added $title to the Movie table.")
    }
}
```

Get an item from a table.

```
suspend fun getMovie(
    tableNameVal: String,
    keyName: String,
    keyVal: String,
) {
    val keyToGet = mutableMapOf<String, AttributeValue>()
```

```
    keyToGet[keyName] = AttributeValue.N(keyVal)
    keyToGet["title"] = AttributeValue.S("King Kong")

    val request =
        GetItemRequest {
            key = keyToGet
            tableName = tableNameVal
        }

    DynamoDbClient.fromEnvironment { region = "us-east-1" }.use { ddb ->
        val returnedItem = ddb.getItem(request)
        val numbersMap = returnedItem.item
        numbersMap?.forEach { key1 ->
            println(key1.key)
            println(key1.value)
        }
    }
}
```

Full example.

```
suspend fun main() {
    val tableName = "Movies"
    val fileName = "../../../resources/sample_files/movies.json"
    val partitionAlias = "#a"

    println("Creating an Amazon DynamoDB table named Movies with a key named id
 and a sort key named title.")
    createScenarioTable(tableName, "year")
    loadData(tableName, fileName)
    getMovie(tableName, "year", "1933")
    scanMovies(tableName)
    val count = queryMovieTable(tableName, "year", partitionAlias)
    println("There are $count Movies released in 2013.")
    deletIssuesTable(tableName)
}

suspend fun createScenarioTable(
    tableNameVal: String,
    key: String,
) {
    val attDef =
```

```
        AttributeDefinition {
            attributeName = key
            attributeType = ScalarAttributeType.N
        }

    val attDef1 =
        AttributeDefinition {
            attributeName = "title"
            attributeType = ScalarAttributeType.S
        }

    val keySchemaVal =
        KeySchemaElement {
            attributeName = key
            keyType = KeyType.Hash
        }

    val keySchemaVal1 =
        KeySchemaElement {
            attributeName = "title"
            keyType = KeyType.Range
        }

    val request =
        CreateTableRequest {
            attributeDefinitions = listOf(attDef, attDef1)
            keySchema = listOf(keySchemaVal, keySchemaVal1)
            billingMode = BillingMode.PayPerRequest
            tableName = tableNameVal
        }

    DynamoDbClient.fromEnvironment { region = "us-east-1" }.use { ddb ->
        val response = ddb.createTable(request)
        ddb.waitUntilTableExists {
            // suspend call
            tableName = tableNameVal
        }
        println("The table was successfully created
 ${response.tableDescription?.tableArn}")
    }
}

// Load data into the table.
suspend fun loadData(
```

```
    tableName: String,
    fileName: String,
) {
    val parser = JsonFactory().createParser(File(fileName))
    val rootNode = ObjectMapper().readTree<JsonNode>(parser)
    val iter: Iterator<JsonNode> = rootNode.iterator()
    var currentNode: ObjectNode

    var t = 0
    while (iter.hasNext()) {
        if (t == 50) {
            break
        }

        currentNode = iter.next() as ObjectNode
        val year = currentNode.path("year").asInt()
        val title = currentNode.path("title").asText()
        val info = currentNode.path("info").toString()
        putMovie(tableName, year, title, info)
        t++
    }
}

suspend fun putMovie(
    tableNameVal: String,
    year: Int,
    title: String,
    info: String,
) {
    val itemValues = mutableMapOf<String, AttributeValue>()
    val strVal = year.toString()
    // Add all content to the table.
    itemValues["year"] = AttributeValue.N(strVal)
    itemValues["title"] = AttributeValue.S(title)
    itemValues["info"] = AttributeValue.S(info)

    val request =
        PutItemRequest {
            tableName = tableNameVal
            item = itemValues
        }

    DynamoDbClient.fromEnvironment { region = "us-east-1" }.use { ddb ->
        ddb.putItem(request)
```

```
        println("Added $title to the Movie table.")
    }
}

suspend fun getMovie(
    tableNameVal: String,
    keyName: String,
    keyVal: String,
) {
    val keyToGet = mutableMapOf<String, AttributeValue>()
    keyToGet[keyName] = AttributeValue.N(keyVal)
    keyToGet["title"] = AttributeValue.S("King Kong")

    val request =
        GetItemRequest {
            key = keyToGet
            tableName = tableNameVal
        }

    DynamoDbClient.fromEnvironment { region = "us-east-1" }.use { ddb ->
        val returnedItem = ddb.getItem(request)
        val numbersMap = returnedItem.item
        numbersMap?.forEach { key1 ->
            println(key1.key)
            println(key1.value)
        }
    }
}

suspend fun deletIssuesTable(tableNameVal: String) {
    val request =
        DeleteTableRequest {
            tableName = tableNameVal
        }

    DynamoDbClient.fromEnvironment { region = "us-east-1" }.use { ddb ->
        ddb.deleteTable(request)
        println("$tableNameVal was deleted")
    }
}

suspend fun queryMovieTable(
    tableNameVal: String,
    partitionKeyName: String,
```

```
    partitionAlias: String,
): Int {
    val attrNameAlias = mutableMapOf<String, String>()
    attrNameAlias[partitionAlias] = "year"

    // Set up mapping of the partition name with the value.
    val attrValues = mutableMapOf<String, AttributeValue>()
    attrValues[":$partitionKeyName"] = AttributeValue.N("2013")

    val request =
        QueryRequest {
            tableName = tableNameVal
            keyConditionExpression = "$partitionAlias = :$partitionKeyName"
            expressionAttributeNames = attrNameAlias
            this.expressionAttributeValues = attrValues
        }

    DynamoDbClient.fromEnvironment { region = "us-east-1" }.use { ddb ->
        val response = ddb.query(request)
        return response.count
    }
}

suspend fun scanMovies(tableNameVal: String) {
    val request =
        ScanRequest {
            tableName = tableNameVal
        }

    DynamoDbClient.fromEnvironment { region = "us-east-1" }.use { ddb ->
        val response = ddb.scan(request)
        response.items?.forEach { item ->
            item.keys.forEach { key ->
                println("The key name is $key\n")
                println("The value is ${item[key]}")
            }
        }
    }
}
```

- For API details, see the following topics in *AWS SDK for Kotlin API reference*.
  - BatchWriteItem

- CreateTable
- DeleteItem
- DeleteTable
- DescribeTable
- GetItem
- PutItem
- Query
- Scan
- UpdateItem

PHP

SDK for PHP

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Because this example uses supporting files, be sure to read the guidance in the PHP examples README.md file.

```
namespace DynamoDb\Basics;

use Aws\DynamoDb\Marshaler;
use DynamoDb;
use DynamoDb\DynamoDBAttribute;
use DynamoDb\DynamoDBService;

use function AwsUtilities\loadMovieData;
use function AwsUtilities\testable_readline;

class GettingStartedWithDynamoDB
{
    public function run()
    {
        echo("\n");
```

```
        echo("--------------------------------------\n");
        print("Welcome to the Amazon DynamoDB getting started demo using PHP!
\n");
        echo("--------------------------------------\n");

        $uuid = uniqid();
        $service = new DynamoDBService();

        $tableName = "ddb_demo_table_$uuid";
        $service->createTable(
            $tableName,
            [
                new DynamoDBAttribute('year', 'N', 'HASH'),
                new DynamoDBAttribute('title', 'S', 'RANGE')
            ]
        );

        echo "Waiting for table...";
        $service->dynamoDbClient->waitUntil("TableExists", ['TableName' =>
 $tableName]);
        echo "table $tableName found!\n";

        echo "What's the name of the last movie you watched?\n";
        while (empty($movieName)) {
            $movieName = testable_readline("Movie name: ");
        }
        echo "And what year was it released?\n";
        $movieYear = "year";
        while (!is_numeric($movieYear) || intval($movieYear) != $movieYear) {
            $movieYear = testable_readline("Year released: ");
        }

        $service->putItem([
            'Item' => [
                'year' => [
                    'N' => "$movieYear",
                ],
                'title' => [
                    'S' => $movieName,
                ],
            ],
            'TableName' => $tableName,
        ]);
```

```
        echo "How would you rate the movie from 1-10?\n";
        $rating = 0;
        while (!is_numeric($rating) || intval($rating) != $rating || $rating < 1
|| $rating > 10) {
            $rating = testable_readline("Rating (1-10): ");
        }
        echo "What was the movie about?\n";
        while (empty($plot)) {
            $plot = testable_readline("Plot summary: ");
        }
        $key = [
            'Item' => [
                'title' => [
                    'S' => $movieName,
                ],
                'year' => [
                    'N' => $movieYear,
                ],
            ]
        ];
        $attributes = ["rating" =>
            [
                'AttributeName' => 'rating',
                'AttributeType' => 'N',
                'Value' => $rating,
            ],
            'plot' => [
                'AttributeName' => 'plot',
                'AttributeType' => 'S',
                'Value' => $plot,
            ]
        ];
        $service->updateItemAttributesByKey($tableName, $key, $attributes);
        echo "Movie added and updated.";

        $batch = json_decode(loadMovieData());

        $service->writeBatch($tableName, $batch);


        $movie = $service->getItemByKey($tableName, $key);
        echo "\nThe movie {$movie['Item']['title']['S']} was released in
{$movie['Item']['year']['N']}.\n";
```

```
        echo "What rating would you like to give {$movie['Item']['title']['S']}?
\n";
        $rating = 0;
        while (!is_numeric($rating) || intval($rating) != $rating || $rating < 1
 || $rating > 10) {
            $rating = testable_readline("Rating (1-10): ");
        }
        $service->updateItemAttributeByKey($tableName, $key, 'rating', 'N',
 $rating);

        $movie = $service->getItemByKey($tableName, $key);
        echo "Ok, you have rated {$movie['Item']['title']['S']} as a
 {$movie['Item']['rating']['N']}\n";

        $service->deleteItemByKey($tableName, $key);
        echo "But, bad news, this was a trap. That movie has now been deleted
 because of your rating...harsh.\n";

        echo "That's okay though. The book was better. Now, for something
 lighter, in what year were you born?\n";
        $birthYear = "not a number";
        while (!is_numeric($birthYear) || $birthYear >= date("Y")) {
            $birthYear = testable_readline("Birth year: ");
        }
        $birthKey = [
            'Key' => [
                'year' => [
                    'N' => "$birthYear",
                ],
            ],
        ];
        $result = $service->query($tableName, $birthKey);
        $marshal = new Marshaler();
        echo "Here are the movies in our collection released the year you were
 born:\n";
        $oops = "Oops! There were no movies released in that year (that we know
 of).\n";
        $display = "";
        foreach ($result['Items'] as $movie) {
            $movie = $marshal->unmarshalItem($movie);
            $display .= $movie['title'] . "\n";
        }
        echo ($display) ?: $oops;
```

```
        $yearsKey = [
            'Key' => [
                'year' => [
                    'N' => [
                        'minRange' => 1990,
                        'maxRange' => 1999,
                    ],
                ],
            ],
        ];
        $filter = "year between 1990 and 1999";
        echo "\nHere's a list of all the movies released in the 90s:\n";
        $result = $service->scan($tableName, $yearsKey, $filter);
        foreach ($result['Items'] as $movie) {
            $movie = $marshal->unmarshalItem($movie);
            echo $movie['title'] . "\n";
        }

        echo "\nCleaning up this demo by deleting table $tableName...\n";
        $service->deleteTable($tableName);
    }
}
```

- For API details, see the following topics in *AWS SDK for PHP API Reference*.
  - BatchWriteItem
  - CreateTable
  - DeleteItem
  - DeleteTable
  - DescribeTable
  - GetItem
  - PutItem
  - Query
  - Scan
  - UpdateItem

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Create a class that encapsulates a DynamoDB table.

```
from decimal import Decimal
from io import BytesIO
import json
import logging
import os
from pprint import pprint
import requests
from zipfile import ZipFile
import boto3
from boto3.dynamodb.conditions import Key
from botocore.exceptions import ClientError
from question import Question

logger = logging.getLogger(__name__)

class Movies:
    """Encapsulates an Amazon DynamoDB table of movie data.

    Example data structure for a movie record in this table:
        {
            "year": 1999,
            "title": "For Love of the Game",
            "info": {
                "directors": ["Sam Raimi"],
                "release_date": "1999-09-15T00:00:00Z",
                "rating": 6.3,
                "plot": "A washed up pitcher flashes through his career.",
                "rank": 4987,
                "running_time_secs": 8220,
                "actors": [
                    "Kevin Costner",
```

```
                "Kelly Preston",
                "John C. Reilly"
            ]
        }
    }
"""

def __init__(self, dyn_resource):
    """
    :param dyn_resource: A Boto3 DynamoDB resource.
    """
    self.dyn_resource = dyn_resource
    # The table variable is set during the scenario in the call to
    # 'exists' if the table exists. Otherwise, it is set by 'create_table'.
    self.table = None


def exists(self, table_name):
    """
    Determines whether a table exists. As a side effect, stores the table in
    a member variable.

    :param table_name: The name of the table to check.
    :return: True when the table exists; otherwise, False.
    """
    try:
        table = self.dyn_resource.Table(table_name)
        table.load()
        exists = True
    except ClientError as err:
        if err.response["Error"]["Code"] == "ResourceNotFoundException":
            exists = False
        else:
            logger.error(
                "Couldn't check for existence of %s. Here's why: %s: %s",
                table_name,
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
    else:
        self.table = table
    return exists
```

```
    def create_table(self, table_name):
        """
        Creates an Amazon DynamoDB table that can be used to store movie data.
        The table uses the release year of the movie as the partition key and the
        title as the sort key.

        :param table_name: The name of the table to create.
        :return: The newly created table.
        """
        try:
            self.table = self.dyn_resource.create_table(
                TableName=table_name,
                KeySchema=[
                    {"AttributeName": "year", "KeyType": "HASH"},  # Partition
key
                    {"AttributeName": "title", "KeyType": "RANGE"},  # Sort key
                ],
                AttributeDefinitions=[
                    {"AttributeName": "year", "AttributeType": "N"},
                    {"AttributeName": "title", "AttributeType": "S"},
                ],
                BillingMode='PAY_PER_REQUEST',
            )
            self.table.wait_until_exists()
        except ClientError as err:
            logger.error(
                "Couldn't create table %s. Here's why: %s: %s",
                table_name,
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
        else:
            return self.table


    def list_tables(self):
        """
        Lists the Amazon DynamoDB tables for the current account.

        :return: The list of tables.
        """
        try:
```

```
            tables = []
            for table in self.dyn_resource.tables.all():
                print(table.name)
                tables.append(table)
        except ClientError as err:
            logger.error(
                "Couldn't list tables. Here's why: %s: %s",
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
        else:
            return tables


    def write_batch(self, movies):
        """
        Fills an Amazon DynamoDB table with the specified data, using the Boto3
        Table.batch_writer() function to put the items in the table.
        Inside the context manager, Table.batch_writer builds a list of
        requests. On exiting the context manager, Table.batch_writer starts
sending
        batches of write requests to Amazon DynamoDB and automatically
        handles chunking, buffering, and retrying.

        :param movies: The data to put in the table. Each item must contain at
least
                       the keys required by the schema that was specified when
the
                       table was created.
        """
        try:
            with self.table.batch_writer() as writer:
                for movie in movies:
                    writer.put_item(Item=movie)
        except ClientError as err:
            logger.error(
                "Couldn't load data into table %s. Here's why: %s: %s",
                self.table.name,
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
```

```
    def add_movie(self, title, year, plot, rating):
        """
        Adds a movie to the table.

        :param title: The title of the movie.
        :param year: The release year of the movie.
        :param plot: The plot summary of the movie.
        :param rating: The quality rating of the movie.
        """
        try:
            self.table.put_item(
                Item={
                    "year": year,
                    "title": title,
                    "info": {"plot": plot, "rating": Decimal(str(rating))},
                }
            )
        except ClientError as err:
            logger.error(
                "Couldn't add movie %s to table %s. Here's why: %s: %s",
                title,
                self.table.name,
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise


    def get_movie(self, title, year):
        """
        Gets movie data from the table for a specific movie.

        :param title: The title of the movie.
        :param year: The release year of the movie.
        :return: The data about the requested movie.
        """
        try:
            response = self.table.get_item(Key={"year": year, "title": title})
        except ClientError as err:
            logger.error(
                "Couldn't get movie %s from table %s. Here's why: %s: %s",
                title,
                self.table.name,
```

```
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
        else:
            return response["Item"]


    def update_movie(self, title, year, rating, plot):
        """
        Updates rating and plot data for a movie in the table.

        :param title: The title of the movie to update.
        :param year: The release year of the movie to update.
        :param rating: The updated rating to the give the movie.
        :param plot: The updated plot summary to give the movie.
        :return: The fields that were updated, with their new values.
        """
        try:
            response = self.table.update_item(
                Key={"year": year, "title": title},
                UpdateExpression="set info.rating=:r, info.plot=:p",
                ExpressionAttributeValues={":r": Decimal(str(rating)), ":p":
plot},
                ReturnValues="UPDATED_NEW",
            )
        except ClientError as err:
            logger.error(
                "Couldn't update movie %s in table %s. Here's why: %s: %s",
                title,
                self.table.name,
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
        else:
            return response["Attributes"]


    def query_movies(self, year):
        """
        Queries for movies that were released in the specified year.

        :param year: The year to query.
```

```
        :return: The list of movies that were released in the specified year.
        """
        try:
            response =
self.table.query(KeyConditionExpression=Key("year").eq(year))
        except ClientError as err:
            logger.error(
                "Couldn't query for movies released in %s. Here's why: %s: %s",
                year,
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
        else:
            return response["Items"]


    def scan_movies(self, year_range):
        """
        Scans for movies that were released in a range of years.
        Uses a projection expression to return a subset of data for each movie.

        :param year_range: The range of years to retrieve.
        :return: The list of movies released in the specified years.
        """
        movies = []
        scan_kwargs = {
            "FilterExpression": Key("year").between(
                year_range["first"], year_range["second"]
            ),
            "ProjectionExpression": "#yr, title, info.rating",
            "ExpressionAttributeNames": {"#yr": "year"},
        }
        try:
            done = False
            start_key = None
            while not done:
                if start_key:
                    scan_kwargs["ExclusiveStartKey"] = start_key
                response = self.table.scan(**scan_kwargs)
                movies.extend(response.get("Items", []))
                start_key = response.get("LastEvaluatedKey", None)
                done = start_key is None
        except ClientError as err:
```

```
            logger.error(
                "Couldn't scan for movies. Here's why: %s: %s",
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise

        return movies


    def delete_movie(self, title, year):
        """
        Deletes a movie from the table.

        :param title: The title of the movie to delete.
        :param year: The release year of the movie to delete.
        """
        try:
            self.table.delete_item(Key={"year": year, "title": title})
        except ClientError as err:
            logger.error(
                "Couldn't delete movie %s. Here's why: %s: %s",
                title,
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise


    def delete_table(self):
        """
        Deletes the table.
        """
        try:
            self.table.delete()
            self.table = None
        except ClientError as err:
            logger.error(
                "Couldn't delete table. Here's why: %s: %s",
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
```

Create a helper function to download and extract the sample JSON file.

```
def get_sample_movie_data(movie_file_name):
    """
    Gets sample movie data, either from a local file or by first downloading it
 from
    the Amazon DynamoDB developer guide.

    :param movie_file_name: The local file name where the movie data is stored in
 JSON format.
    :return: The movie data as a dict.
    """
    if not os.path.isfile(movie_file_name):
        print(f"Downloading {movie_file_name}...")
        movie_content = requests.get(
            "https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/
samples/moviedata.zip"
        )
        movie_zip = ZipFile(BytesIO(movie_content.content))
        movie_zip.extractall()

    try:
        with open(movie_file_name) as movie_file:
            movie_data = json.load(movie_file, parse_float=Decimal)
    except FileNotFoundError:
        print(
            f"File {movie_file_name} not found. You must first download the file
 to "
            "run this demo. See the README for instructions."
        )
        raise
    else:
        # The sample file lists over 4000 movies, return only the first 250.
        return movie_data[:250]
```

Run an interactive scenario to create the table and perform actions on it.

```
def run_scenario(table_name, movie_file_name, dyn_resource):
    logging.basicConfig(level=logging.INFO, format="%(levelname)s: %(message)s")

    print("-" * 88)
    print("Welcome to the Amazon DynamoDB getting started demo.")
    print("-" * 88)

    movies = Movies(dyn_resource)
    movies_exists = movies.exists(table_name)
    if not movies_exists:
        print(f"\nCreating table {table_name}...")
        movies.create_table(table_name)
        print(f"\nCreated table {movies.table.name}.")

    my_movie = Question.ask_questions(
        [
            Question(
                "title", "Enter the title of a movie you want to add to the
 table: "
            ),
            Question("year", "What year was it released? ", Question.is_int),
            Question(
                "rating",
                "On a scale of 1 - 10, how do you rate it? ",
                Question.is_float,
                Question.in_range(1, 10),
            ),
            Question("plot", "Summarize the plot for me: "),
        ]
    )
    movies.add_movie(**my_movie)
    print(f"\nAdded '{my_movie['title']}' to '{movies.table.name}'.")
    print("-" * 88)

    movie_update = Question.ask_questions(
        [
            Question(
                "rating",
                f"\nLet's update your movie.\nYou rated it {my_movie['rating']},
 what new "
                f"rating would you give it? ",
                Question.is_float,
                Question.in_range(1, 10),
```

```
            ),
            Question(
                "plot",
                f"You summarized the plot as '{my_movie['plot']}'.\nWhat would
you say now? ",
            ),
        ]
    )
    my_movie.update(movie_update)
    updated = movies.update_movie(**my_movie)
    print(f"\nUpdated '{my_movie['title']}' with new attributes:")
    pprint(updated)
    print("-" * 88)

    if not movies_exists:
        movie_data = get_sample_movie_data(movie_file_name)
        print(f"\nReading data from '{movie_file_name}' into your table.")
        movies.write_batch(movie_data)
        print(f"\nWrote {len(movie_data)} movies into {movies.table.name}.")
    print("-" * 88)

    title = "The Lord of the Rings: The Fellowship of the Ring"
    if Question.ask_question(
        f"Let's move on...do you want to get info about '{title}'? (y/n) ",
        Question.is_yesno,
    ):
        movie = movies.get_movie(title, 2001)
        print("\nHere's what I found:")
        pprint(movie)
    print("-" * 88)

    ask_for_year = True
    while ask_for_year:
        release_year = Question.ask_question(
            f"\nLet's get a list of movies released in a given year. Enter a year
between "
            f"1972 and 2018: ",
            Question.is_int,
            Question.in_range(1972, 2018),
        )
        releases = movies.query_movies(release_year)
        if releases:
            print(f"There were {len(releases)} movies released in
{release_year}:")
```

```
            for release in releases:
                print(f"\t{release['title']}")
            ask_for_year = False
        else:
            print(f"I don't know about any movies released in {release_year}!")
            ask_for_year = Question.ask_question(
                "Try another year? (y/n) ", Question.is_yesno
            )
    print("-" * 88)

    years = Question.ask_questions(
        [
            Question(
                "first",
                f"\nNow let's scan for movies released in a range of years. Enter
a year: ",
                Question.is_int,
                Question.in_range(1972, 2018),
            ),
            Question(
                "second",
                "Now enter another year: ",
                Question.is_int,
                Question.in_range(1972, 2018),
            ),
        ]
    )
    releases = movies.scan_movies(years)
    if releases:
        count = Question.ask_question(
            f"\nFound {len(releases)} movies. How many do you want to see? ",
            Question.is_int,
            Question.in_range(1, len(releases)),
        )
        print(f"\nHere are your {count} movies:\n")
        pprint(releases[:count])
    else:
        print(
            f"I don't know about any movies released between {years['first']} "
            f"and {years['second']}."
        )
    print("-" * 88)

    if Question.ask_question(
```

```
        f"\nLet's remove your movie from the table. Do you want to remove "
        f"'{my_movie['title']}'? (y/n)",
        Question.is_yesno,
    ):
        movies.delete_movie(my_movie["title"], my_movie["year"])
        print(f"\nRemoved '{my_movie['title']}' from the table.")
    print("-" * 88)

    if Question.ask_question(f"\nDelete the table? (y/n) ", Question.is_yesno):
        movies.delete_table()
        print(f"Deleted {table_name}.")
    else:
        print(
            "Don't forget to delete the table when you're done or you might incur
 "
            "charges on your account."
        )

    print("\nThanks for watching!")
    print("-" * 88)


if __name__ == "__main__":
    try:
        run_scenario(
            "doc-example-table-movies", "moviedata.json",
 boto3.resource("dynamodb")
        )
    except Exception as e:
        print(f"Something went wrong with the demo! Here's what: {e}")
```

This scenario uses the following helper class to ask questions at a command prompt.

```
class Question:
    """
    A helper class to ask questions at a command prompt and validate and convert
    the answers.
    """

    def __init__(self, key, question, *validators):
        """
        :param key: The key that is used for storing the answer in a dict, when
```

```
                    multiple questions are asked in a set.
        :param question: The question to ask.
        :param validators: The answer is passed through the list of validators
until
                           one fails or they all pass. Validators may also
convert the
                           answer to another form, such as from a str to an int.
        """
        self.key = key
        self.question = question
        self.validators = Question.non_empty, *validators

    @staticmethod
    def ask_questions(questions):
        """
        Asks a set of questions and stores the answers in a dict.

        :param questions: The list of questions to ask.
        :return: A dict of answers.
        """
        answers = {}
        for question in questions:
            answers[question.key] = Question.ask_question(
                question.question, *question.validators
            )
        return answers

    @staticmethod
    def ask_question(question, *validators):
        """
        Asks a single question and validates it against a list of validators.
        When an answer fails validation, the complaint is printed and the
question
        is asked again.

        :param question: The question to ask.
        :param validators: The list of validators that the answer must pass.
        :return: The answer, converted to its final form by the validators.
        """
        answer = None
        while answer is None:
            answer = input(question)
            for validator in validators:
                answer, complaint = validator(answer)
```

```
                if answer is None:
                    print(complaint)
                    break
        return answer

    @staticmethod
    def non_empty(answer):
        """
        Validates that the answer is not empty.
        :return: The non-empty answer, or None.
        """
        return answer if answer != "" else None, "I need an answer. Please?"

    @staticmethod
    def is_yesno(answer):
        """
        Validates a yes/no answer.
        :return: True when the answer is 'y'; otherwise, False.
        """
        return answer.lower() == "y", ""

    @staticmethod
    def is_int(answer):
        """
        Validates that the answer can be converted to an int.
        :return: The int answer; otherwise, None.
        """
        try:
            int_answer = int(answer)
        except ValueError:
            int_answer = None
        return int_answer, f"{answer} must be a valid integer."

    @staticmethod
    def is_letter(answer):
        """
        Validates that the answer is a letter.
        :return The letter answer, converted to uppercase; otherwise, None.
        """
        return (
            answer.upper() if answer.isalpha() else None,
            f"{answer} must be a single letter.",
        )
```

```
    @staticmethod
    def is_float(answer):
        """
        Validate that the answer can be converted to a float.
        :return The float answer; otherwise, None.
        """
        try:
            float_answer = float(answer)
        except ValueError:
            float_answer = None
        return float_answer, f"{answer} must be a valid float."

    @staticmethod
    def in_range(lower, upper):
        """
        Validate that the answer is within a range. The answer must be of a type
that can
        be compared to the lower and upper bounds.
        :return: The answer, if it is within the range; otherwise, None.
        """

        def _validate(answer):
            return (
                answer if lower <= answer <= upper else None,
                f"{answer} must be between {lower} and {upper}.",
            )

        return _validate
```

- For API details, see the following topics in *AWS SDK for Python (Boto3) API Reference*.
  - BatchWriteItem
  - CreateTable
  - DeleteItem
  - DeleteTable
  - DescribeTable
  - GetItem
  - PutItem

- Query
- Scan
- UpdateItem

Ruby

SDK for Ruby

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Create a class that encapsulates a DynamoDB table.

```
# Creates an Amazon DynamoDB table that can be used to store movie data.
# The table uses the release year of the movie as the partition key and the
# title as the sort key.
#
# @param table_name [String] The name of the table to create.
# @return [Aws::DynamoDB::Table] The newly created table.
def create_table(table_name)
  @table = @dynamo_resource.create_table(
    table_name: table_name,
    key_schema: [
      { attribute_name: 'year', key_type: 'HASH' }, # Partition key
      { attribute_name: 'title', key_type: 'RANGE' } # Sort key
    ],
    attribute_definitions: [
      { attribute_name: 'year', attribute_type: 'N' },
      { attribute_name: 'title', attribute_type: 'S' }
    ],
    billing_mode: 'PAY_PER_REQUEST'
  )
  @dynamo_resource.client.wait_until(:table_exists, table_name: table_name)
  @table
rescue Aws::DynamoDB::Errors::ServiceError => e
  @logger.error("Failed create table #{table_name}:\n#{e.code}: #{e.message}")
  raise
end
```

Create a helper function to download and extract the sample JSON file.

```
  # Gets sample movie data, either from a local file or by first downloading it
 from
  # the Amazon DynamoDB Developer Guide.
  #
  # @param movie_file_name [String] The local file name where the movie data is
 stored in JSON format.
  # @return [Hash] The movie data as a Hash.
  def fetch_movie_data(movie_file_name)
    if !File.file?(movie_file_name)
      @logger.debug("Downloading #{movie_file_name}...")
      movie_content = URI.open(
        'https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/
samples/moviedata.zip'
      )
      movie_json = ''
      Zip::File.open_buffer(movie_content) do |zip|
        zip.each do |entry|
          movie_json = entry.get_input_stream.read
        end
      end
    else
      movie_json = File.read(movie_file_name)
    end
    movie_data = JSON.parse(movie_json)
    # The sample file lists over 4000 movies. This returns only the first 250.
    movie_data.slice(0, 250)
  rescue StandardError => e
    puts("Failure downloading movie data:\n#{e}")
    raise
  end
```

Run an interactive scenario to create the table and perform actions on it.

```
  table_name = "doc-example-table-movies-#{rand(10**4)}"
  scaffold = Scaffold.new(table_name)
  dynamodb_wrapper = DynamoDBBasics.new(table_name)

  new_step(1, 'Create a new DynamoDB table if none already exists.')
```

```
 unless scaffold.exists?(table_name)
   puts("\nNo such table: #{table_name}. Creating it...")
   scaffold.create_table(table_name)
   print "Done!\n".green
 end

 new_step(2, 'Add a new record to the DynamoDB table.')
 my_movie = {}
 my_movie[:title] = CLI::UI::Prompt.ask('Enter the title of a movie to add to
the table. E.g. The Matrix')
 my_movie[:year] = CLI::UI::Prompt.ask('What year was it released? E.g.
1989').to_i
 my_movie[:rating] = CLI::UI::Prompt.ask('On a scale of 1 - 10, how do you rate
it? E.g. 7').to_i
 my_movie[:plot] = CLI::UI::Prompt.ask('Enter a brief summary of the plot. E.g.
A man awakens to a new reality.')
 dynamodb_wrapper.add_item(my_movie)
 puts("\nNew record added:")
 puts JSON.pretty_generate(my_movie).green
 print "Done!\n".green

 new_step(3, 'Update a record in the DynamoDB table.')
 my_movie[:rating] = CLI::UI::Prompt.ask("Let's update the movie you added with
a new rating, e.g. 3:").to_i
 response = dynamodb_wrapper.update_item(my_movie)
 puts("Updated '#{my_movie[:title]}' with new attributes:")
 puts JSON.pretty_generate(response).green
 print "Done!\n".green

 new_step(4, 'Get a record from the DynamoDB table.')
 puts("Searching for #{my_movie[:title]} (#{my_movie[:year]})...")
 response = dynamodb_wrapper.get_item(my_movie[:title], my_movie[:year])
 puts JSON.pretty_generate(response).green
 print "Done!\n".green

 new_step(5, 'Write a batch of items into the DynamoDB table.')
 download_file = 'moviedata.json'
 puts("Downloading movie database to #{download_file}...")
 movie_data = scaffold.fetch_movie_data(download_file)
 puts("Writing movie data from #{download_file} into your table...")
 scaffold.write_batch(movie_data)
 puts("Records added: #{movie_data.length}.")
 print "Done!\n".green
```

```
 new_step(5, 'Query for a batch of items by key.')
 loop do
   release_year = CLI::UI::Prompt.ask('Enter a year between 1972 and 2018, e.g.
1999:').to_i
   results = dynamodb_wrapper.query_items(release_year)
   if results.any?
     puts("There were #{results.length} movies released in #{release_year}:")
     results.each do |movie|
       print "\t #{movie['title']}".green
     end
     break
   else
     continue = CLI::UI::Prompt.ask("Found no movies released in
#{release_year}! Try another year? (y/n)")
     break unless continue.eql?('y')
   end
 end
 print "\nDone!\n".green

 new_step(6, 'Scan for a batch of items using a filter expression.')
 years = {}
 years[:start] = CLI::UI::Prompt.ask('Enter a starting year between 1972 and
2018:')
 years[:end] = CLI::UI::Prompt.ask('Enter an ending year between 1972 and
2018:')
 releases = dynamodb_wrapper.scan_items(years)
 if !releases.empty?
   puts("Found #{releases.length} movies.")
   count = Question.ask(
     'How many do you want to see? ', method(:is_int), in_range(1,
releases.length)
   )
   puts("Here are your #{count} movies:")
   releases.take(count).each do |release|
     puts("\t#{release['title']}")
   end
 else
   puts("I don't know about any movies released between #{years[:start]} "\
        "and #{years[:end]}.")
 end
 print "\nDone!\n".green

 new_step(7, 'Delete an item from the DynamoDB table.')
```

```
  answer = CLI::UI::Prompt.ask("Do you want to remove '#{my_movie[:title]}'? (y/
n) ")
  if answer.eql?('y')
    dynamodb_wrapper.delete_item(my_movie[:title], my_movie[:year])
    puts("Removed '#{my_movie[:title]}' from the table.")
    print "\nDone!\n".green
  end

  new_step(8, 'Delete the DynamoDB table.')
  answer = CLI::UI::Prompt.ask('Delete the table? (y/n)')
  if answer.eql?('y')
    scaffold.delete_table
    puts("Deleted #{table_name}.")
  else
    puts("Don't forget to delete the table when you're done!")
  end
  print "\nThanks for watching!\n".green
rescue Aws::Errors::ServiceError
  puts('Something went wrong with the demo.')
rescue Errno::ENOENT
  true
end
```

- For API details, see the following topics in *AWS SDK for Ruby API Reference*.
  - BatchWriteItem
  - CreateTable
  - DeleteItem
  - DeleteTable
  - DescribeTable
  - GetItem
  - PutItem
  - Query
  - Scan
  - UpdateItem

SAP ABAP

SDK for SAP ABAP

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    " Create an Amazon Dynamo DB table.

    TRY.
        DATA(lo_session) = /aws1/cl_rt_session_aws=>create( cv_pfl ).
        DATA(lo_dyn) = /aws1/cl_dyn_factory=>create( lo_session ).
        DATA(lt_keyschema) = VALUE /aws1/cl_dynkeyschemaelement=>tt_keyschema(
          ( NEW /aws1/cl_dynkeyschemaelement( iv_attributename = 'year'
                                              iv_keytype = 'HASH' ) )
          ( NEW /aws1/cl_dynkeyschemaelement( iv_attributename = 'title'
                                              iv_keytype = 'RANGE' ) ) ).
        DATA(lt_attributedefinitions) = VALUE /aws1/
cl_dynattributedefn=>tt_attributedefinitions(
          ( NEW /aws1/cl_dynattributedefn( iv_attributename = 'year'
                                           iv_attributetype = 'N' ) )
          ( NEW /aws1/cl_dynattributedefn( iv_attributename = 'title'
                                           iv_attributetype = 'S' ) ) ).

        " Adjust read/write capacities as desired.
        DATA(lo_dynprovthroughput)  = NEW /aws1/cl_dynprovthroughput(
          iv_readcapacityunits = 5
          iv_writecapacityunits = 5 ).
        DATA(oo_result) = lo_dyn->createtable(
          it_keyschema = lt_keyschema
          iv_tablename = iv_table_name
          it_attributedefinitions = lt_attributedefinitions
          io_provisionedthroughput = lo_dynprovthroughput ).
        " Table creation can take some time. Wait till table exists before
 returning.
        lo_dyn->get_waiter( )->tableexists(
          iv_max_wait_time = 200
          iv_tablename     = iv_table_name ).
        MESSAGE 'DynamoDB Table' && iv_table_name && 'created.' TYPE 'I'.
```

```
        " It throws exception if the table already exists.
      CATCH /aws1/cx_dynresourceinuseex INTO DATA(lo_resourceinuseex).
        DATA(lv_error) = |"{ lo_resourceinuseex->av_err_code }" -
 { lo_resourceinuseex->av_err_msg }|.
        MESSAGE lv_error TYPE 'E'.
    ENDTRY.

    " Describe table
    TRY.
        DATA(lo_table) = lo_dyn->describetable( iv_tablename = iv_table_name ).
        DATA(lv_tablename) = lo_table->get_table( )->ask_tablename( ).
        MESSAGE 'The table name is ' && lv_tablename TYPE 'I'.
      CATCH /aws1/cx_dynresourcenotfoundex.
        MESSAGE 'The table does not exist' TYPE 'E'.
    ENDTRY.

    " Put items into the table.
    TRY.
        DATA(lo_resp_putitem) = lo_dyn->putitem(
          iv_tablename = iv_table_name
          it_item      = VALUE /aws1/
cl_dynattributevalue=>tt_putiteminputattributemap(
            ( VALUE /aws1/cl_dynattributevalue=>ts_putiteminputattrmap_maprow(
              key = 'title' value = NEW /aws1/cl_dynattributevalue( iv_s =
 'Jaws' ) ) )
            ( VALUE /aws1/cl_dynattributevalue=>ts_putiteminputattrmap_maprow(
              key = 'year' value = NEW /aws1/cl_dynattributevalue( iv_n = |
{ '1975' }| ) ) )
            ( VALUE /aws1/cl_dynattributevalue=>ts_putiteminputattrmap_maprow(
              key = 'rating' value = NEW /aws1/cl_dynattributevalue( iv_n = |
{ '7.5' }| ) ) )
          ) ).
        lo_resp_putitem = lo_dyn->putitem(
          iv_tablename = iv_table_name
          it_item      = VALUE /aws1/
cl_dynattributevalue=>tt_putiteminputattributemap(
            ( VALUE /aws1/cl_dynattributevalue=>ts_putiteminputattrmap_maprow(
              key = 'title' value = NEW /aws1/cl_dynattributevalue( iv_s = 'Star
 Wars' ) ) )
            ( VALUE /aws1/cl_dynattributevalue=>ts_putiteminputattrmap_maprow(
              key = 'year' value = NEW /aws1/cl_dynattributevalue( iv_n = |
{ '1978' }| ) ) )
            ( VALUE /aws1/cl_dynattributevalue=>ts_putiteminputattrmap_maprow(
```

```
              key = 'rating' value = NEW /aws1/cl_dynattributevalue( iv_n = |
{ '8.1' }| ) ) )
          ) ).
        lo_resp_putitem = lo_dyn->putitem(
          iv_tablename = iv_table_name
          it_item      = VALUE /aws1/
cl_dynattributevalue=>tt_putiteminputattributemap(
            ( VALUE /aws1/cl_dynattributevalue=>ts_putiteminputattrmap_maprow(
              key = 'title' value = NEW /aws1/cl_dynattributevalue( iv_s =
 'Speed' ) ) )
            ( VALUE /aws1/cl_dynattributevalue=>ts_putiteminputattrmap_maprow(
              key = 'year' value = NEW /aws1/cl_dynattributevalue( iv_n = |
{ '1994' }| ) ) )
            ( VALUE /aws1/cl_dynattributevalue=>ts_putiteminputattrmap_maprow(
              key = 'rating' value = NEW /aws1/cl_dynattributevalue( iv_n = |
{ '7.9' }| ) ) )
          ) ).
        " TYPE REF TO /AWSEX/CL_AWS1_dyn_PUT_ITEM_OUTPUT
        MESSAGE '3 rows inserted into DynamoDB Table' && iv_table_name TYPE 'I'.
      CATCH /aws1/cx_dyncondalcheckfaile00.
        MESSAGE 'A condition specified in the operation could not be evaluated.'
 TYPE 'E'.
      CATCH /aws1/cx_dynresourcenotfoundex.
        MESSAGE 'The table or index does not exist' TYPE 'E'.
      CATCH /aws1/cx_dyntransactconflictex.
        MESSAGE 'Another transaction is using the item' TYPE 'E'.
    ENDTRY.

    " Get item from table.
    TRY.
        DATA(lo_resp_getitem) = lo_dyn->getitem(
          iv_tablename                = iv_table_name
          it_key                      = VALUE /aws1/cl_dynattributevalue=>tt_key(
           ( VALUE /aws1/cl_dynattributevalue=>ts_key_maprow(
             key = 'title' value = NEW /aws1/cl_dynattributevalue( iv_s =
 'Jaws' ) ) )
           ( VALUE /aws1/cl_dynattributevalue=>ts_key_maprow(
             key = 'year' value = NEW /aws1/cl_dynattributevalue( iv_n =
 '1975' ) ) )
          ) ).
        DATA(lt_attr) = lo_resp_getitem->get_item( ).
        DATA(lo_title) = lt_attr[ key = 'title' ]-value.
        DATA(lo_year) = lt_attr[ key = 'year' ]-value.
        DATA(lo_rating) = lt_attr[ key = 'year' ]-value.
```

```
        MESSAGE 'Movie name is: ' && lo_title->get_s( ) TYPE 'I'.
        MESSAGE 'Movie year is: ' && lo_year->get_n( ) TYPE 'I'.
        MESSAGE 'Movie rating is: ' && lo_rating->get_n( ) TYPE 'I'.
      CATCH /aws1/cx_dynresourcenotfoundex.
        MESSAGE 'The table or index does not exist' TYPE 'E'.
    ENDTRY.

    " Query item from table.
    TRY.
        DATA(lt_attributelist) = VALUE /aws1/
cl_dynattributevalue=>tt_attributevaluelist(
             ( NEW /aws1/cl_dynattributevalue( iv_n = '1975' ) ) ).
        DATA(lt_keyconditions) = VALUE /aws1/cl_dyncondition=>tt_keyconditions(
          ( VALUE /aws1/cl_dyncondition=>ts_keyconditions_maprow(
          key = 'year'
          value = NEW /aws1/cl_dyncondition(
            it_attributevaluelist = lt_attributelist
            iv_comparisonoperator = |EQ|
          ) ) ) ).
        DATA(lo_query_result) = lo_dyn->query(
          iv_tablename = iv_table_name
          it_keyconditions = lt_keyconditions ).
        DATA(lt_items) = lo_query_result->get_items( ).
        READ TABLE lo_query_result->get_items( ) INTO DATA(lt_item) INDEX 1.
        lo_title = lt_item[ key = 'title' ]-value.
        lo_year = lt_item[ key = 'year' ]-value.
        lo_rating = lt_item[ key = 'rating' ]-value.
        MESSAGE 'Movie name is: ' && lo_title->get_s( ) TYPE 'I'.
        MESSAGE 'Movie year is: ' && lo_year->get_n( ) TYPE 'I'.
        MESSAGE 'Movie rating is: ' && lo_rating->get_n( ) TYPE 'I'.
      CATCH /aws1/cx_dynresourcenotfoundex.
        MESSAGE 'The table or index does not exist' TYPE 'E'.
    ENDTRY.

    " Scan items from table.
    TRY.
        DATA(lo_scan_result) = lo_dyn->scan( iv_tablename = iv_table_name ).
        lt_items = lo_scan_result->get_items( ).
        " Read the first item and display the attributes.
        READ TABLE lo_query_result->get_items( ) INTO lt_item INDEX 1.
        lo_title = lt_item[ key = 'title' ]-value.
        lo_year = lt_item[ key = 'year' ]-value.
        lo_rating = lt_item[ key = 'rating' ]-value.
        MESSAGE 'Movie name is: ' && lo_title->get_s( ) TYPE 'I'.
```

```
        MESSAGE 'Movie year is: ' && lo_year->get_n( ) TYPE 'I'.
        MESSAGE 'Movie rating is: ' && lo_rating->get_n( ) TYPE 'I'.
      CATCH /aws1/cx_dynresourcenotfoundex.
        MESSAGE 'The table or index does not exist' TYPE 'E'.
    ENDTRY.

    " Update items from table.
    TRY.
        DATA(lt_attributeupdates) = VALUE /aws1/
cl_dynattrvalueupdate=>tt_attributeupdates(
          ( VALUE /aws1/cl_dynattrvalueupdate=>ts_attributeupdates_maprow(
          key = 'rating' value = NEW /aws1/cl_dynattrvalueupdate(
            io_value  = NEW /aws1/cl_dynattributevalue( iv_n = '7.6' )
            iv_action = |PUT| ) ) ) ).
        DATA(lt_key) = VALUE /aws1/cl_dynattributevalue=>tt_key(
          ( VALUE /aws1/cl_dynattributevalue=>ts_key_maprow(
            key = 'year' value = NEW /aws1/cl_dynattributevalue( iv_n =
 '1975' ) ) )
          ( VALUE /aws1/cl_dynattributevalue=>ts_key_maprow(
            key = 'title' value = NEW /aws1/cl_dynattributevalue( iv_s =
 '1980' ) ) ) ).
        DATA(lo_resp) = lo_dyn->updateitem(
          iv_tablename        = iv_table_name
          it_key              = lt_key
          it_attributeupdates = lt_attributeupdates ).
        MESSAGE '1 item updated in DynamoDB Table' && iv_table_name TYPE 'I'.
      CATCH /aws1/cx_dyncondalcheckfaile00.
        MESSAGE 'A condition specified in the operation could not be evaluated.'
 TYPE 'E'.
      CATCH /aws1/cx_dynresourcenotfoundex.
        MESSAGE 'The table or index does not exist' TYPE 'E'.
      CATCH /aws1/cx_dyntransactconflictex.
        MESSAGE 'Another transaction is using the item' TYPE 'E'.
    ENDTRY.

    " Delete table.
    TRY.
        lo_dyn->deletetable( iv_tablename = iv_table_name ).
        lo_dyn->get_waiter( )->tablenotexists(
          iv_max_wait_time = 200
          iv_tablename     = iv_table_name ).
        MESSAGE 'DynamoDB Table deleted.' TYPE 'I'.
      CATCH /aws1/cx_dynresourcenotfoundex.
        MESSAGE 'The table or index does not exist' TYPE 'E'.
```

```
      CATCH /aws1/cx_dynresourceinuseex.
        MESSAGE 'The table cannot be deleted as it is in use' TYPE 'E'.
    ENDTRY.
```

- For API details, see the following topics in *AWS SDK for SAP ABAP API reference*.
  - BatchWriteItem
  - CreateTable
  - DeleteItem
  - DeleteTable
  - DescribeTable
  - GetItem
  - PutItem
  - Query
  - Scan
  - UpdateItem

Swift

SDK for Swift

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

A Swift class that handles DynamoDB calls to the SDK for Swift.

```
import AWSDynamoDB
import Foundation

/// An enumeration of error codes representing issues that can arise when using
/// the `MovieTable` class.
enum MoviesError: Error {
    /// The specified table wasn't found or couldn't be created.
    case TableNotFound
```

```
    /// The specified item wasn't found or couldn't be created.
    case ItemNotFound
    /// The Amazon DynamoDB client is not properly initialized.
    case UninitializedClient
    /// The table status reported by Amazon DynamoDB is not recognized.
    case StatusUnknown
    /// One or more specified attribute values are invalid or missing.
    case InvalidAttributes
}

/// A class representing an Amazon DynamoDB table containing movie
/// information.
public class MovieTable {
    var ddbClient: DynamoDBClient?
    let tableName: String

    /// Create an object representing a movie table in an Amazon DynamoDB
    /// database.
    ///
    /// - Parameters:
    ///   - region: The optional Amazon Region to create the database in.
    ///   - tableName: The name to assign to the table. If not specified, a
    ///     random table name is generated automatically.
    ///
    /// > Note: The table is not necessarily available when this function
    /// returns. Use `tableExists()` to check for its availability, or
    /// `awaitTableActive()` to wait until the table's status is reported as
    /// ready to use by Amazon DynamoDB.
    ///
    init(region: String? = nil, tableName: String) async throws {
        do {
            let config = try await DynamoDBClient.DynamoDBClientConfiguration()
            if let region = region {
                config.region = region
            }

            self.ddbClient = DynamoDBClient(config: config)
            self.tableName = tableName

            try await self.createTable()
        } catch {
            print("ERROR: ", dump(error, name: "Initializing Amazon
 DynamoDBClient client"))
            throw error
```

```
        }
    }

    ///
    /// Create a movie table in the Amazon DynamoDB data store.
    ///
    private func createTable() async throws {
        do {
            guard let client = self.ddbClient else {
                throw MoviesError.UninitializedClient
            }

            let input = CreateTableInput(
                attributeDefinitions: [
                    DynamoDBClientTypes.AttributeDefinition(attributeName:
"year", attributeType: .n),
                    DynamoDBClientTypes.AttributeDefinition(attributeName:
"title", attributeType: .s)
                ],
                billingMode: DynamoDBClientTypes.BillingMode.payPerRequest,
                keySchema: [
                    DynamoDBClientTypes.KeySchemaElement(attributeName: "year",
keyType: .hash),
                    DynamoDBClientTypes.KeySchemaElement(attributeName: "title",
keyType: .range)
                ],
                tableName: self.tableName
            )
            let output = try await client.createTable(input: input)
            if output.tableDescription == nil {
                throw MoviesError.TableNotFound
            }
        } catch {
            print("ERROR: createTable:", dump(error))
            throw error
        }
    }


    /// Check to see if the table exists online yet.
    ///
    /// - Returns: `true` if the table exists, or `false` if not.
    ///
    func tableExists() async throws -> Bool {
```

```
        do {
            guard let client = self.ddbClient else {
                throw MoviesError.UninitializedClient
            }

            let input = DescribeTableInput(
                tableName: tableName
            )
            let output = try await client.describeTable(input: input)
            guard let description = output.table else {
                throw MoviesError.TableNotFound
            }

            return description.tableName == self.tableName
        } catch {
            print("ERROR: tableExists:", dump(error))
            throw error
        }
    }


    ///
    /// Waits for the table to exist and for its status to be active.
    ///
    func awaitTableActive() async throws {
        while try (await self.tableExists() == false) {
            do {
                let duration = UInt64(0.25 * 1_000_000_000) // Convert .25
seconds to nanoseconds.
                try await Task.sleep(nanoseconds: duration)
            } catch {
                print("Sleep error:", dump(error))
            }
        }

        while try (await self.getTableStatus() != .active) {
            do {
                let duration = UInt64(0.25 * 1_000_000_000) // Convert .25
seconds to nanoseconds.
                try await Task.sleep(nanoseconds: duration)
            } catch {
                print("Sleep error:", dump(error))
            }
        }
```

```
    }


    ///
    /// Deletes the table from Amazon DynamoDB.
    ///
    func deleteTable() async throws {
        do {
            guard let client = self.ddbClient else {
                throw MoviesError.UninitializedClient
            }

            let input = DeleteTableInput(
                tableName: self.tableName
            )
            _ = try await client.deleteTable(input: input)
        } catch {
            print("ERROR: deleteTable:", dump(error))
            throw error
        }
    }


    /// Get the table's status.
    ///
    /// - Returns: The table status, as defined by the
    ///   `DynamoDBClientTypes.TableStatus` enum.
    ///
    func getTableStatus() async throws -> DynamoDBClientTypes.TableStatus {
        do {
            guard let client = self.ddbClient else {
                throw MoviesError.UninitializedClient
            }

            let input = DescribeTableInput(
                tableName: self.tableName
            )
            let output = try await client.describeTable(input: input)
            guard let description = output.table else {
                throw MoviesError.TableNotFound
            }
            guard let status = description.tableStatus else {
                throw MoviesError.StatusUnknown
            }
```

```
            return status
        } catch {
            print("ERROR: getTableStatus:", dump(error))
            throw error
        }
    }


    /// Populate the movie database from the specified JSON file.
    ///
    /// - Parameter jsonPath: Path to a JSON file containing movie data.
    ///
    func populate(jsonPath: String) async throws {
        do {
            guard let client = self.ddbClient else {
                throw MoviesError.UninitializedClient
            }

            // Create a Swift `URL` and use it to load the file into a `Data`
            // object. Then decode the JSON into an array of `Movie` objects.

            let fileUrl = URL(fileURLWithPath: jsonPath)
            let jsonData = try Data(contentsOf: fileUrl)

            var movieList = try JSONDecoder().decode([Movie].self, from:
jsonData)

            // Truncate the list to the first 200 entries or so for this example.

            if movieList.count > 200 {
                movieList = Array(movieList[...199])
            }

            // Before sending records to the database, break the movie list into
            // 25-entry chunks, which is the maximum size of a batch item
request.

            let count = movieList.count
            let chunks = stride(from: 0, to: count, by: 25).map {
                Array(movieList[$0 ..< Swift.min($0 + 25, count)])
            }

            // For each chunk, create a list of write request records and
populate
```

```
            // them with `PutRequest` requests, each specifying one movie from
the
            // chunk. Once the chunk's items are all in the `PutRequest` list,
            // send them to Amazon DynamoDB using the
            // `DynamoDBClient.batchWriteItem()` function.

            for chunk in chunks {
                var requestList: [DynamoDBClientTypes.WriteRequest] = []

                for movie in chunk {
                    let item = try await movie.getAsItem()
                    let request = DynamoDBClientTypes.WriteRequest(
                        putRequest: .init(
                            item: item
                        )
                    )
                    requestList.append(request)
                }

                let input = BatchWriteItemInput(requestItems: [tableName:
requestList])
                _ = try await client.batchWriteItem(input: input)
            }
        } catch {
            print("ERROR: populate:", dump(error))
            throw error
        }
    }


    /// Add a movie specified as a `Movie` structure to the Amazon DynamoDB
    /// table.
    ///
    /// - Parameter movie: The `Movie` to add to the table.
    ///
    func add(movie: Movie) async throws {
        do {
            guard let client = self.ddbClient else {
                throw MoviesError.UninitializedClient
            }

            // Get a DynamoDB item containing the movie data.
            let item = try await movie.getAsItem()
```

```
            // Send the `PutItem` request to Amazon DynamoDB.

            let input = PutItemInput(
                item: item,
                tableName: self.tableName
            )
            _ = try await client.putItem(input: input)
        } catch {
            print("ERROR: add movie:", dump(error))
            throw error
        }
    }


    /// Given a movie's details, add a movie to the Amazon DynamoDB table.
    ///
    /// - Parameters:
    ///   - title: The movie's title as a `String`.
    ///   - year: The release year of the movie (`Int`).
    ///   - rating: The movie's rating if available (`Double`; default is
    ///     `nil`).
    ///   - plot: A summary of the movie's plot (`String`; default is `nil`,
    ///     indicating no plot summary is available).
    ///
    func add(title: String, year: Int, rating: Double? = nil,
             plot: String? = nil) async throws
    {
        do {
            let movie = Movie(title: title, year: year, rating: rating, plot:
plot)
            try await self.add(movie: movie)
        } catch {
            print("ERROR: add with fields:", dump(error))
            throw error
        }
    }


    /// Return a `Movie` record describing the specified movie from the Amazon
    /// DynamoDB table.
    ///
    /// - Parameters:
    ///   - title: The movie's title (`String`).
    ///   - year: The movie's release year (`Int`).
```

```
    ///
    /// - Throws: `MoviesError.ItemNotFound` if the movie isn't in the table.
    ///
    /// - Returns: A `Movie` record with the movie's details.
    func get(title: String, year: Int) async throws -> Movie {
        do {
            guard let client = self.ddbClient else {
                throw MoviesError.UninitializedClient
            }

            let input = GetItemInput(
                key: [
                    "year": .n(String(year)),
                    "title": .s(title)
                ],
                tableName: self.tableName
            )
            let output = try await client.getItem(input: input)
            guard let item = output.item else {
                throw MoviesError.ItemNotFound
            }

            let movie = try Movie(withItem: item)
            return movie
        } catch {
            print("ERROR: get:", dump(error))
            throw error
        }
    }


    /// Get all the movies released in the specified year.
    ///
    /// - Parameter year: The release year of the movies to return.
    ///
    /// - Returns: An array of `Movie` objects describing each matching movie.
    ///
    func getMovies(fromYear year: Int) async throws -> [Movie] {
        do {
            guard let client = self.ddbClient else {
                throw MoviesError.UninitializedClient
            }

            let input = QueryInput(
```

```
                expressionAttributeNames: [
                    "#y": "year"
                ],
                expressionAttributeValues: [
                    ":y": .n(String(year))
                ],
                keyConditionExpression: "#y = :y",
                tableName: self.tableName
            )
            // Use "Paginated" to get all the movies.
            // This lets the SDK handle the 'lastEvaluatedKey' property in
"QueryOutput".

            let pages = client.queryPaginated(input: input)

            var movieList: [Movie] = []
            for try await page in pages {
                guard let items = page.items else {
                    print("Error: no items returned.")
                    continue
                }

                // Convert the found movies into `Movie` objects and return an
array
                // of them.

                for item in items {
                    let movie = try Movie(withItem: item)
                    movieList.append(movie)
                }
            }
            return movieList
        } catch {
            print("ERROR: getMovies:", dump(error))
            throw error
        }
    }


    /// Return an array of `Movie` objects released in the specified range of
    /// years.
    ///
    /// - Parameters:
    ///   - firstYear: The first year of movies to return.
```

```
    ///   - lastYear: The last year of movies to return.
    ///   - startKey: A starting point to resume processing; always use `nil`.
    ///
    /// - Returns: An array of `Movie` objects describing the matching movies.
    ///
    /// > Note: The `startKey` parameter is used by this function when
    ///   recursively calling itself, and should always be `nil` when calling
    ///   directly.
    ///
    func getMovies(firstYear: Int, lastYear: Int,
                   startKey: [Swift.String: DynamoDBClientTypes.AttributeValue]?
= nil)
        async throws -> [Movie]
    {
        do {
            var movieList: [Movie] = []

            guard let client = self.ddbClient else {
                throw MoviesError.UninitializedClient
            }

            let input = ScanInput(
                consistentRead: true,
                exclusiveStartKey: startKey,
                expressionAttributeNames: [
                    "#y": "year" // `year` is a reserved word, so use `#y`
instead.
                ],
                expressionAttributeValues: [
                    ":y1": .n(String(firstYear)),
                    ":y2": .n(String(lastYear))
                ],
                filterExpression: "#y BETWEEN :y1 AND :y2",
                tableName: self.tableName
            )

            let pages = client.scanPaginated(input: input)

            for try await page in pages {
                guard let items = page.items else {
                    print("Error: no items returned.")
                    continue
                }
```

```
                // Build an array of `Movie` objects for the returned items.

                for item in items {
                    let movie = try Movie(withItem: item)
                    movieList.append(movie)
                }
            }
            return movieList

        } catch {
            print("ERROR: getMovies with scan:", dump(error))
            throw error
        }
    }


    /// Update the specified movie with new `rating` and `plot` information.
    ///
    /// - Parameters:
    ///   - title: The title of the movie to update.
    ///   - year: The release year of the movie to update.
    ///   - rating: The new rating for the movie.
    ///   - plot: The new plot summary string for the movie.
    ///
    /// - Returns: An array of mappings of attribute names to their new
    ///   listing each item actually changed. Items that didn't need to change
    ///   aren't included in this list. `nil` if no changes were made.
    ///
    func update(title: String, year: Int, rating: Double? = nil, plot: String? =
nil) async throws
        -> [Swift.String: DynamoDBClientTypes.AttributeValue]?
    {
        do {
            guard let client = self.ddbClient else {
                throw MoviesError.UninitializedClient
            }

            // Build the update expression and the list of expression attribute
            // values. Include only the information that's changed.

            var expressionParts: [String] = []
            var attrValues: [Swift.String: DynamoDBClientTypes.AttributeValue] =
[:]
```

```
            if rating != nil {
                expressionParts.append("info.rating=:r")
                attrValues[":r"] = .n(String(rating!))
            }
            if plot != nil {
                expressionParts.append("info.plot=:p")
                attrValues[":p"] = .s(plot!)
            }
            let expression = "set \(expressionParts.joined(separator: ", "))"

            let input = UpdateItemInput(
                // Create substitution tokens for the attribute values, to ensure
                // no conflicts in expression syntax.
                expressionAttributeValues: attrValues,
                // The key identifying the movie to update consists of the
release
                // year and title.
                key: [
                    "year": .n(String(year)),
                    "title": .s(title)
                ],
                returnValues: .updatedNew,
                tableName: self.tableName,
                updateExpression: expression
            )
            let output = try await client.updateItem(input: input)

            guard let attributes: [Swift.String:
DynamoDBClientTypes.AttributeValue] = output.attributes else {
                throw MoviesError.InvalidAttributes
            }
            return attributes
        } catch {
            print("ERROR: update:", dump(error))
            throw error
        }
    }


    /// Delete a movie, given its title and release year.
    ///
    /// - Parameters:
    ///   - title: The movie's title.
    ///   - year: The movie's release year.
```

```
    ///
    func delete(title: String, year: Int) async throws {
        do {
            guard let client = self.ddbClient else {
                throw MoviesError.UninitializedClient
            }

            let input = DeleteItemInput(
                key: [
                    "year": .n(String(year)),
                    "title": .s(title)
                ],
                tableName: self.tableName
            )
            _ = try await client.deleteItem(input: input)
        } catch {
            print("ERROR: delete:", dump(error))
            throw error
        }
    }
}
```

The structures used by the MovieTable class to represent movies.

```
import Foundation
import AWSDynamoDB

/// The optional details about a movie.
public struct Details: Codable {
    /// The movie's rating, if available.
    var rating: Double?
    /// The movie's plot, if available.
    var plot: String?
}

/// A structure describing a movie. The `year` and `title` properties are
/// required and are used as the key for Amazon DynamoDB operations. The
/// `info` sub-structure's two properties, `rating` and `plot`, are optional.
public struct Movie: Codable {
    /// The year in which the movie was released.
    var year: Int
```

```
    /// The movie's title.
    var title: String
    /// A `Details` object providing the optional movie rating and plot
    /// information.
    var info: Details

    /// Create a `Movie` object representing a movie, given the movie's
    /// details.
    ///
    /// - Parameters:
    ///   - title: The movie's title (`String`).
    ///   - year: The year in which the movie was released (`Int`).
    ///   - rating: The movie's rating (optional `Double`).
    ///   - plot: The movie's plot (optional `String`)
    init(title: String, year: Int, rating: Double? = nil, plot: String? = nil) {
        self.title = title
        self.year = year

        self.info = Details(rating: rating, plot: plot)
    }

    /// Create a `Movie` object representing a movie, given the movie's
    /// details.
    ///
    /// - Parameters:
    ///   - title: The movie's title (`String`).
    ///   - year: The year in which the movie was released (`Int`).
    ///   - info: The optional rating and plot information for the movie in a
    ///     `Details` object.
    init(title: String, year: Int, info: Details?){
        self.title = title
        self.year = year

        if info != nil {
            self.info = info!
        } else {
            self.info = Details(rating: nil, plot: nil)
        }
    }

    ///
    /// Return a new `MovieTable` object, given an array mapping string to Amazon
    /// DynamoDB attribute values.
    ///
```

```
    /// - Parameter item: The item information provided to the form used by
    ///   DynamoDB. This is an array of strings mapped to
    ///   `DynamoDBClientTypes.AttributeValue` values.
    init(withItem item: [Swift.String:DynamoDBClientTypes.AttributeValue]) throws
  {
        // Read the attributes.

        guard let titleAttr = item["title"],
              let yearAttr = item["year"] else {
            throw MoviesError.ItemNotFound
        }
        let infoAttr = item["info"] ?? nil

        // Extract the values of the title and year attributes.

        if case .s(let titleVal) = titleAttr {
            self.title = titleVal
        } else {
            throw MoviesError.InvalidAttributes
        }

        if case .n(let yearVal) = yearAttr {
            self.year = Int(yearVal)!
        } else {
            throw MoviesError.InvalidAttributes
        }

        // Extract the rating and/or plot from the `info` attribute, if
        // they're present.

        var rating: Double? = nil
        var plot: String? = nil

        if infoAttr != nil, case .m(let infoVal) = infoAttr {
            let ratingAttr = infoVal["rating"] ?? nil
            let plotAttr = infoVal["plot"] ?? nil

            if ratingAttr != nil, case .n(let ratingVal) = ratingAttr {
                rating = Double(ratingVal) ?? nil
            }
            if plotAttr != nil, case .s(let plotVal) = plotAttr {
                plot = plotVal
            }
        }
```

```
        self.info = Details(rating: rating, plot: plot)
    }

    ///
    /// Return an array mapping attribute names to Amazon DynamoDB attribute
    /// values, representing the contents of the `Movie` record as a DynamoDB
    /// item.
    ///
    /// - Returns: The movie item as an array of type
    ///   `[Swift.String:DynamoDBClientTypes.AttributeValue]`.
    ///
    func getAsItem() async throws ->
[Swift.String:DynamoDBClientTypes.AttributeValue]  {
        // Build the item record, starting with the year and title, which are
        // always present.

        var item: [Swift.String:DynamoDBClientTypes.AttributeValue] = [
            "year": .n(String(self.year)),
            "title": .s(self.title)
        ]

        // Add the `info` field with the rating and/or plot if they're
        // available.

        var details: [Swift.String:DynamoDBClientTypes.AttributeValue] = [:]
        if (self.info.rating != nil || self.info.plot != nil) {
            if self.info.rating != nil {
                details["rating"] = .n(String(self.info.rating!))
            }
            if self.info.plot != nil {
                details["plot"] = .s(self.info.plot!)
            }
        }
        item["info"] = .m(details)

        return item
    }
}
```

A program that uses the MovieTable class to access a DynamoDB database.

```
import ArgumentParser
import ClientRuntime
import Foundation

import AWSDynamoDB


@testable import MovieList

extension String {
    // Get the directory if the string is a file path.
    func directory() -> String {
        guard let lastIndex = lastIndex(of: "/") else {
            print("Error: String directory separator not found.")
            return ""
        }
        return String(self[...lastIndex])
    }
}

struct ExampleCommand: ParsableCommand {
    @Argument(help: "The path of the sample movie data JSON file.")
    var jsonPath: String = #file.directory() + "../../../../../resources/
sample_files/movies.json"

    @Option(help: "The AWS Region to run AWS API calls in.")
    var awsRegion: String?

    @Option(
        help: ArgumentHelp("The level of logging for the Swift SDK to perform."),
        completion: .list([
            "critical",
            "debug",
            "error",
            "info",
            "notice",
            "trace",
            "warning"
        ])
    )
    var logLevel: String = "error"

    /// Configuration details for the command.
```

```
    static var configuration = CommandConfiguration(
        commandName: "basics",
        abstract: "A basic scenario demonstrating the usage of Amazon DynamoDB.",
        discussion: """
        An example showing how to use Amazon DynamoDB to perform a series of
        common database activities on a simple movie database.
        """
    )

    /// Called by ``main()`` to asynchronously run the AWS example.
    func runAsync() async throws {
        print("Welcome to the AWS SDK for Swift basic scenario for Amazon
DynamoDB!")

        //========================================================================
        // 1. Create the table. The Amazon DynamoDB table is represented by
        //    the `MovieTable` class.
        //========================================================================

        let tableName = "ddb-movies-sample-\(Int.random(in: 1 ... Int.max))"

        print("Creating table \"\(tableName)\"...")

        let movieDatabase = try await MovieTable(region: awsRegion,
                                                 tableName: tableName)

        print("\nWaiting for table to be ready to use...")
        try await movieDatabase.awaitTableActive()

        //========================================================================
        // 2. Add a movie to the table.
        //========================================================================

        print("\nAdding a movie...")
        try await movieDatabase.add(title: "Avatar: The Way of Water", year:
2022)
        try await movieDatabase.add(title: "Not a Real Movie", year: 2023)

        //========================================================================
        // 3. Update the plot and rating of the movie using an update
        //    expression.
        //========================================================================

        print("\nAdding details to the added movie...")
```

```
        _ = try await movieDatabase.update(title: "Avatar: The Way of Water",
year: 2022,
                                           rating: 9.2, plot: "It's a sequel.")

        //===========================================================================
        // 4. Populate the table from the JSON file.
        //===========================================================================

        print("\nPopulating the movie database from JSON...")
        try await movieDatabase.populate(jsonPath: jsonPath)

        //===========================================================================
        // 5. Get a specific movie by key. In this example, the key is a
        //    combination of `title` and `year`.
        //===========================================================================

        print("\nLooking for a movie in the table...")
        let gotMovie = try await movieDatabase.get(title: "This Is the End",
year: 2013)

        print("Found the movie \"\(gotMovie.title)\", released in
\(gotMovie.year).")
        print("Rating: \(gotMovie.info.rating ?? 0.0).")
        print("Plot summary: \(gotMovie.info.plot ?? "None.")")

        //===========================================================================
        // 6. Delete a movie.
        //===========================================================================

        print("\nDeleting the added movie...")
        try await movieDatabase.delete(title: "Avatar: The Way of Water", year:
2022)

        //===========================================================================
        // 7. Use a query with a key condition expression to return all movies
        //    released in a given year.
        //===========================================================================

        print("\nGetting movies released in 1994...")
        let movieList = try await movieDatabase.getMovies(fromYear: 1994)
        for movie in movieList {
            print("    \(movie.title)")
        }
```

```
        //====================================================================
        // 8. Use `scan()` to return movies released in a range of years.
        //====================================================================

        print("\nGetting movies released between 1993 and 1997...")
        let scannedMovies = try await movieDatabase.getMovies(firstYear: 1993,
 lastYear: 1997)
        for movie in scannedMovies {
            print("    \(movie.title) (\(movie.year))")
        }

        //====================================================================
        // 9. Delete the table.
        //====================================================================

        print("\nDeleting the table...")
        try await movieDatabase.deleteTable()
    }
}

@main
struct Main {
    static func main() async {
        let args = Array(CommandLine.arguments.dropFirst())

        do {
            let command = try ExampleCommand.parse(args)
            try await command.runAsync()
        } catch {
            ExampleCommand.exit(withError: error)
        }
    }
}
```

- For API details, see the following topics in *AWS SDK for Swift API reference*.
  - BatchWriteItem
  - CreateTable
  - DeleteItem
  - DeleteTable

- DescribeTable
- GetItem
- PutItem
- Query
- Scan
- UpdateItem

For a complete list of AWS SDK developer guides and code examples, see Using DynamoDB with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Actions for DynamoDB using AWS SDKs

The following code examples demonstrate how to perform individual DynamoDB actions with AWS SDKs. Each example includes a link to GitHub, where you can find instructions for setting up and running the code.

These excerpts call the DynamoDB API and are code excerpts from larger programs that must be run in context. You can see actions in context in Scenarios for DynamoDB using AWS SDKs.

The following examples include only the most commonly used actions. For a complete list, see the Amazon DynamoDB API Reference.

Examples

- Use BatchExecuteStatement with an AWS SDK
- Use BatchGetItem with an AWS SDK or CLI
- Use BatchWriteItem with an AWS SDK or CLI
- Use CreateTable with an AWS SDK or CLI
- Use DeleteItem with an AWS SDK or CLI
- Use DeleteTable with an AWS SDK or CLI
- Use DescribeTable with an AWS SDK or CLI
- Use DescribeTimeToLive with an AWS SDK or CLI
- Use ExecuteStatement with an AWS SDK
- Use GetItem with an AWS SDK or CLI
- Use ListTables with an AWS SDK or CLI

- Use PutItem with an AWS SDK or CLI
- Use Query with an AWS SDK or CLI
- Use Scan with an AWS SDK or CLI
- Use UpdateItem with an AWS SDK or CLI
- Use UpdateTable with an AWS SDK or CLI
- Use UpdateTimeToLive with an AWS SDK or CLI

Use `BatchExecuteStatement` with an AWS SDK

The following code examples show how to use `BatchExecuteStatement`.

Action examples are code excerpts from larger programs and must be run in context. You can see this action in context in the following code examples:

- Delete data using PartiQL DELETE
- Insert data using PartiQL INSERT
- Query a table by using batches of PartiQL statements
- Query data using PartiQL SELECT
- Update data using PartiQL UPDATE

.NET

SDK for .NET

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Use batches of INSERT statements to add items.

```
        /// <summary>
        /// Inserts movies imported from a JSON file into the movie table by
        /// using an Amazon DynamoDB PartiQL INSERT statement.
        /// </summary>
        /// <param name="tableName">The name of the table into which the movie
```

```
        /// information will be inserted.</param>
        /// <param name="movieFileName">The name of the JSON file that contains
        /// movie information.</param>
        /// <returns>A Boolean value that indicates the success or failure of
        /// the insert operation.</returns>
        public static async Task<bool> InsertMovies(string tableName, string
movieFileName)
        {
            // Get the list of movies from the JSON file.
            var movies = ImportMovies(movieFileName);

            var success = false;

            if (movies is not null)
            {
                // Insert the movies in a batch using PartiQL. Because the
                // batch can contain a maximum of 25 items, insert 25 movies
                // at a time.
                string insertBatch = $"INSERT INTO {tableName} VALUE
{{'title': ?, 'year': ?}}";
                var statements = new List<BatchStatementRequest>();

                try
                {
                    for (var indexOffset = 0; indexOffset < 250; indexOffset +=
25)
                    {
                        for (var i = indexOffset; i < indexOffset + 25; i++)
                        {
                            statements.Add(new BatchStatementRequest
                            {
                                Statement = insertBatch,
                                Parameters = new List<AttributeValue>
                                {
                                    new AttributeValue { S = movies[i].Title },
                                    new AttributeValue { N =
movies[i].Year.ToString() },
                                },
                            });
                        }

                        var response = await
Client.BatchExecuteStatementAsync(new BatchExecuteStatementRequest
                        {
```

```
                            Statements = statements,
                        });

                        // Wait between batches for movies to be successfully
added.
                        System.Threading.Thread.Sleep(3000);

                        success = response.HttpStatusCode ==
System.Net.HttpStatusCode.OK;

                        // Clear the list of statements for the next batch.
                        statements.Clear();
                    }
                }
                catch (AmazonDynamoDBException ex)
                {
                    Console.WriteLine(ex.Message);
                }
            }

            return success;
        }

        /// <summary>
        /// Loads the contents of a JSON file into a list of movies to be
        /// added to the DynamoDB table.
        /// </summary>
        /// <param name="movieFileName">The full path to the JSON file.</param>
        /// <returns>A generic list of movie objects.</returns>
        public static List<Movie> ImportMovies(string movieFileName)
        {
            if (!File.Exists(movieFileName))
            {
                return null!;
            }

            using var sr = new StreamReader(movieFileName);
            string json = sr.ReadToEnd();
            var allMovies = JsonConvert.DeserializeObject<List<Movie>>(json);

            if (allMovies is not null)
            {
                // Return the first 250 entries.
                return allMovies.GetRange(0, 250);
```

```
            }
            else
            {
                return null!;
            }
        }
```

Use batches of SELECT statements to get items.

```
        /// <summary>
        /// Gets movies from the movie table by
        /// using an Amazon DynamoDB PartiQL SELECT statement.
        /// </summary>
        /// <param name="tableName">The name of the table.</param>
        /// <param name="title1">The title of the first movie.</param>
        /// <param name="title2">The title of the second movie.</param>
        /// <param name="year1">The year of the first movie.</param>
        /// <param name="year2">The year of the second movie.</param>
        /// <returns>True if successful.</returns>
        public static async Task<bool> GetBatch(
            string tableName,
            string title1,
            string title2,
            int year1,
            int year2)
        {
            var getBatch = $"SELECT * FROM {tableName} WHERE title = ? AND year
= ?";
            var statements = new List<BatchStatementRequest>
            {
                new BatchStatementRequest
                {
                    Statement = getBatch,
                    Parameters = new List<AttributeValue>
                    {
                        new AttributeValue { S = title1 },
                        new AttributeValue { N = year1.ToString() },
                    },
                },

                new BatchStatementRequest
                {
```

```
                    Statement = getBatch,
                    Parameters = new List<AttributeValue>
                    {
                        new AttributeValue { S = title2 },
                        new AttributeValue { N = year2.ToString() },
                    },
                }
            };

            var response = await Client.BatchExecuteStatementAsync(new
BatchExecuteStatementRequest
            {
                Statements = statements,
            });

            if (response.Responses.Count > 0)
            {
                response.Responses.ForEach(r =>
                {
                    if (r.Item.Any())
                    {

Console.WriteLine($"{r.Item["title"]}\t{r.Item["year"]}");
                    }
                });
                return true;
            }
            else
            {
                Console.WriteLine($"Couldn't find either {title1} or {title2}.");
                return false;
            }

        }
```

Use batches of UPDATE statements to update items.

```
        /// <summary>
        /// Updates information for multiple movies.
        /// </summary>
        /// <param name="tableName">The name of the table containing the
        /// movies to be updated.</param>
```

```
        /// <param name="producer1">The producer name for the first movie
        /// to update.</param>
        /// <param name="title1">The title of the first movie.</param>
        /// <param name="year1">The year that the first movie was released.</
param>
        /// <param name="producer2">The producer name for the second
        /// movie to update.</param>
        /// <param name="title2">The title of the second movie.</param>
        /// <param name="year2">The year that the second movie was released.</
param>
        /// <returns>A Boolean value that indicates the success of the update.</
returns>
        public static async Task<bool> UpdateBatch(
            string tableName,
            string producer1,
            string title1,
            int year1,
            string producer2,
            string title2,
            int year2)
        {

            string updateBatch = $"UPDATE {tableName} SET Producer=? WHERE title
 = ? AND year = ?";
            var statements = new List<BatchStatementRequest>
            {
                new BatchStatementRequest
                {
                    Statement = updateBatch,
                    Parameters = new List<AttributeValue>
                    {
                        new AttributeValue { S = producer1 },
                        new AttributeValue { S = title1 },
                        new AttributeValue { N = year1.ToString() },
                    },
                },

                new BatchStatementRequest
                {
                    Statement = updateBatch,
                    Parameters = new List<AttributeValue>
                    {
                        new AttributeValue { S = producer2 },
                        new AttributeValue { S = title2 },
```

```
                        new AttributeValue { N = year2.ToString() },
                    },
                }
            };

            var response = await Client.BatchExecuteStatementAsync(new
 BatchExecuteStatementRequest
            {
                Statements = statements,
            });

            return response.HttpStatusCode == System.Net.HttpStatusCode.OK;
        }
```

Use batches of DELETE statements to delete items.

```
        /// <summary>
        /// Deletes multiple movies using a PartiQL BatchExecuteAsync
        /// statement.
        /// </summary>
        /// <param name="tableName">The name of the table containing the
        /// moves that will be deleted.</param>
        /// <param name="title1">The title of the first movie.</param>
        /// <param name="year1">The year the first movie was released.</param>
        /// <param name="title2">The title of the second movie.</param>
        /// <param name="year2">The year the second movie was released.</param>
        /// <returns>A Boolean value indicating the success of the operation.</
returns>
        public static async Task<bool> DeleteBatch(
            string tableName,
            string title1,
            int year1,
            string title2,
            int year2)
        {

            string updateBatch = $"DELETE FROM {tableName} WHERE title = ? AND
 year = ?";
            var statements = new List<BatchStatementRequest>
            {
                new BatchStatementRequest
                {
```

```
                    Statement = updateBatch,
                    Parameters = new List<AttributeValue>
                    {
                        new AttributeValue { S = title1 },
                        new AttributeValue { N = year1.ToString() },
                    },
                },

                new BatchStatementRequest
                {
                    Statement = updateBatch,
                    Parameters = new List<AttributeValue>
                    {
                        new AttributeValue { S = title2 },
                        new AttributeValue { N = year2.ToString() },
                    },
                }
            };

            var response = await Client.BatchExecuteStatementAsync(new
BatchExecuteStatementRequest
            {
                Statements = statements,
            });

            return response.HttpStatusCode == System.Net.HttpStatusCode.OK;
        }
```

- For API details, see BatchExecuteStatement in *AWS SDK for .NET API Reference*.

C++

SDK for C++

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Use batches of INSERT statements to add items.

```
    // 2. Add multiple movies using "Insert" statements. (BatchExecuteStatement)
    Aws::DynamoDB::DynamoDBClient dynamoClient(clientConfiguration);

    std::vector<Aws::String> titles;
    std::vector<float> ratings;
    std::vector<int> years;
    std::vector<Aws::String> plots;
    Aws::String doAgain = "n";
    do {
        Aws::String aTitle = askQuestion(
                "Enter the title of a movie you want to add to the table: ");
        titles.push_back(aTitle);
        int aYear = askQuestionForInt("What year was it released? ");
        years.push_back(aYear);
        float aRating = askQuestionForFloatRange(
                "On a scale of 1 - 10, how do you rate it? ",
                1, 10);
        ratings.push_back(aRating);
        Aws::String aPlot = askQuestion("Summarize the plot for me: ");
        plots.push_back(aPlot);

        doAgain = askQuestion(Aws::String("Would you like to add more movies? (y/
n) "));
    } while (doAgain == "y");

    std::cout << "Adding " << titles.size()
              << (titles.size() == 1 ? " movie " : " movies ")
              << "to the table using a batch \"INSERT\" statement." << std::endl;

    {
        Aws::Vector<Aws::DynamoDB::Model::BatchStatementRequest> statements(
                titles.size());

        std::stringstream sqlStream;
        sqlStream << "INSERT INTO \"" << MOVIE_TABLE_NAME << "\" VALUE {'"
                  << TITLE_KEY << "': ?, '" << YEAR_KEY << "': ?, '"
                  << INFO_KEY << "': ?}";

        std::string sql(sqlStream.str());

        for (size_t i = 0; i < statements.size(); ++i) {
            statements[i].SetStatement(sql);
```

```
            Aws::Vector<Aws::DynamoDB::Model::AttributeValue> attributes;
            attributes.push_back(
                    Aws::DynamoDB::Model::AttributeValue().SetS(titles[i]));

attributes.push_back(Aws::DynamoDB::Model::AttributeValue().SetN(years[i]));

            // Create attribute for the info map.
            Aws::DynamoDB::Model::AttributeValue infoMapAttribute;

            std::shared_ptr<Aws::DynamoDB::Model::AttributeValue> ratingAttribute
= Aws::MakeShared<Aws::DynamoDB::Model::AttributeValue>(
                    ALLOCATION_TAG.c_str());
            ratingAttribute->SetN(ratings[i]);
            infoMapAttribute.AddMEntry(RATING_KEY, ratingAttribute);

            std::shared_ptr<Aws::DynamoDB::Model::AttributeValue> plotAttribute =
Aws::MakeShared<Aws::DynamoDB::Model::AttributeValue>(
                    ALLOCATION_TAG.c_str());
            plotAttribute->SetS(plots[i]);
            infoMapAttribute.AddMEntry(PLOT_KEY, plotAttribute);
            attributes.push_back(infoMapAttribute);
            statements[i].SetParameters(attributes);
        }

        Aws::DynamoDB::Model::BatchExecuteStatementRequest request;

        request.SetStatements(statements);

        Aws::DynamoDB::Model::BatchExecuteStatementOutcome outcome =
dynamoClient.BatchExecuteStatement(
                request);
        if (!outcome.IsSuccess()) {
            std::cerr << "Failed to add the movies: " <<
outcome.GetError().GetMessage()
                      << std::endl;
            return false;
        }
    }
```

Use batches of SELECT statements to get items.

```
   // 3. Get the data for multiple movies using "Select" statements.
(BatchExecuteStatement)
   {
       Aws::Vector<Aws::DynamoDB::Model::BatchStatementRequest> statements(
               titles.size());
       std::stringstream sqlStream;
       sqlStream << "SELECT * FROM  \"" << MOVIE_TABLE_NAME << "\" WHERE "
                 << TITLE_KEY << "=? and " << YEAR_KEY << "=?";

       std::string sql(sqlStream.str());

       for (size_t i = 0; i < statements.size(); ++i) {
           statements[i].SetStatement(sql);
           Aws::Vector<Aws::DynamoDB::Model::AttributeValue> attributes;
           attributes.push_back(
                   Aws::DynamoDB::Model::AttributeValue().SetS(titles[i]));

attributes.push_back(Aws::DynamoDB::Model::AttributeValue().SetN(years[i]));
           statements[i].SetParameters(attributes);
       }

       Aws::DynamoDB::Model::BatchExecuteStatementRequest request;

       request.SetStatements(statements);

       Aws::DynamoDB::Model::BatchExecuteStatementOutcome outcome =
dynamoClient.BatchExecuteStatement(
               request);
       if (outcome.IsSuccess()) {
           const Aws::DynamoDB::Model::BatchExecuteStatementResult &result =
outcome.GetResult();

           const Aws::Vector<Aws::DynamoDB::Model::BatchStatementResponse>
&responses = result.GetResponses();

           for (const Aws::DynamoDB::Model::BatchStatementResponse &response:
responses) {
               const Aws::Map<Aws::String, Aws::DynamoDB::Model::AttributeValue>
&item = response.GetItem();

               printMovieInfo(item);
           }
       }
```

```
        else {
            std::cerr << "Failed to retrieve the movie information: "
                      << outcome.GetError().GetMessage() << std::endl;
            return false;
        }
    }
```

Use batches of UPDATE statements to update items.

```
    // 4. Update the data for multiple movies using "Update" statements.
(BatchExecuteStatement)

    for (size_t i = 0; i < titles.size(); ++i) {
        ratings[i] = askQuestionForFloatRange(
                Aws::String("\nLet's update your the movie, \"") + titles[i] +
                ".\nYou rated it  " + std::to_string(ratings[i])
                + ", what new rating would you give it? ", 1, 10);
    }

    std::cout << "Updating the movie with a batch \"UPDATE\" statement." <<
std::endl;

    {
        Aws::Vector<Aws::DynamoDB::Model::BatchStatementRequest> statements(
                titles.size());

        std::stringstream sqlStream;
        sqlStream << "UPDATE \"" << MOVIE_TABLE_NAME << "\" SET "
                  << INFO_KEY << "." << RATING_KEY << "=? WHERE "
                  << TITLE_KEY << "=? AND " << YEAR_KEY << "=?";


        std::string sql(sqlStream.str());

        for (size_t i = 0; i < statements.size(); ++i) {
            statements[i].SetStatement(sql);

            Aws::Vector<Aws::DynamoDB::Model::AttributeValue> attributes;
            attributes.push_back(
                    Aws::DynamoDB::Model::AttributeValue().SetN(ratings[i]));
            attributes.push_back(
                    Aws::DynamoDB::Model::AttributeValue().SetS(titles[i]));
```

```
attributes.push_back(Aws::DynamoDB::Model::AttributeValue().SetN(years[i]));
            statements[i].SetParameters(attributes);
        }

        Aws::DynamoDB::Model::BatchExecuteStatementRequest request;

        request.SetStatements(statements);
        Aws::DynamoDB::Model::BatchExecuteStatementOutcome outcome =
dynamoClient.BatchExecuteStatement(
                request);
        if (!outcome.IsSuccess()) {
            std::cerr << "Failed to update movie information: "
                      << outcome.GetError().GetMessage() << std::endl;
            return false;
        }
    }
```

Use batches of DELETE statements to delete items.

```
    // 6. Delete multiple movies using "Delete" statements.
(BatchExecuteStatement)
    {
        Aws::Vector<Aws::DynamoDB::Model::BatchStatementRequest> statements(
                titles.size());
        std::stringstream sqlStream;
        sqlStream << "DELETE FROM  \"" << MOVIE_TABLE_NAME << "\" WHERE "
                  << TITLE_KEY << "=? and " << YEAR_KEY << "=?";

        std::string sql(sqlStream.str());

        for (size_t i = 0; i < statements.size(); ++i) {
            statements[i].SetStatement(sql);
            Aws::Vector<Aws::DynamoDB::Model::AttributeValue> attributes;
            attributes.push_back(
                    Aws::DynamoDB::Model::AttributeValue().SetS(titles[i]));

attributes.push_back(Aws::DynamoDB::Model::AttributeValue().SetN(years[i]));
            statements[i].SetParameters(attributes);
        }

        Aws::DynamoDB::Model::BatchExecuteStatementRequest request;
```

```
        request.SetStatements(statements);

        Aws::DynamoDB::Model::BatchExecuteStatementOutcome outcome =
 dynamoClient.BatchExecuteStatement(
                request);

        if (!outcome.IsSuccess()) {
            std::cerr << "Failed to delete the movies: "
                      << outcome.GetError().GetMessage() << std::endl;
            return false;
        }
    }
```

- For API details, see BatchExecuteStatement in *AWS SDK for C++ API Reference*.

Go

SDK for Go V2

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Define a function receiver struct for the example.

```
import (
 "context"
 "fmt"
 "log"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/attributevalue"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb/types"
)

// PartiQLRunner encapsulates the Amazon DynamoDB service actions used in the
```

```
// PartiQL examples. It contains a DynamoDB service client that is used to act on
 the
// specified table.
type PartiQLRunner struct {
 DynamoDbClient *dynamodb.Client
 TableName      string
}
```

Use batches of INSERT statements to add items.

```
// AddMovieBatch runs a batch of PartiQL INSERT statements to add multiple movies
 to the
// DynamoDB table.
func (runner PartiQLRunner) AddMovieBatch(ctx context.Context, movies []Movie)
 error {
 statementRequests := make([]types.BatchStatementRequest, len(movies))
 for index, movie := range movies {
  params, err := attributevalue.MarshalList([]interface{}{movie.Title,
 movie.Year, movie.Info})
  if err != nil {
   panic(err)
  }
  statementRequests[index] = types.BatchStatementRequest{
   Statement: aws.String(fmt.Sprintf(
    "INSERT INTO \"%v\" VALUE {'title': ?, 'year': ?, 'info': ?}",
 runner.TableName)),
   Parameters: params,
  }
 }

 _, err := runner.DynamoDbClient.BatchExecuteStatement(ctx,
 &dynamodb.BatchExecuteStatementInput{
  Statements: statementRequests,
 })
 if err != nil {
  log.Printf("Couldn't insert a batch of items with PartiQL. Here's why: %v\n",
 err)
 }
 return err
}
```

Use batches of SELECT statements to get items.

```
// GetMovieBatch runs a batch of PartiQL SELECT statements to get multiple movies
 from
// the DynamoDB table by title and year.
func (runner PartiQLRunner) GetMovieBatch(ctx context.Context, movies []Movie)
 ([]Movie, error) {
 statementRequests := make([]types.BatchStatementRequest, len(movies))
 for index, movie := range movies {
  params, err := attributevalue.MarshalList([]interface{}{movie.Title,
 movie.Year})
  if err != nil {
   panic(err)
  }
  statementRequests[index] = types.BatchStatementRequest{
   Statement: aws.String(
    fmt.Sprintf("SELECT * FROM \"%v\" WHERE title=? AND year=?",
 runner.TableName)),
   Parameters: params,
  }
 }

 output, err := runner.DynamoDbClient.BatchExecuteStatement(ctx,
 &dynamodb.BatchExecuteStatementInput{
  Statements: statementRequests,
 })
 var outMovies []Movie
 if err != nil {
  log.Printf("Couldn't get a batch of items with PartiQL. Here's why: %v\n", err)
 } else {
  for _, response := range output.Responses {
   var movie Movie
   err = attributevalue.UnmarshalMap(response.Item, &movie)
   if err != nil {
    log.Printf("Couldn't unmarshal response. Here's why: %v\n", err)
   } else {
    outMovies = append(outMovies, movie)
   }
  }
```

```
 }
 return outMovies, err
}
```

Use batches of UPDATE statements to update items.

```
// UpdateMovieBatch runs a batch of PartiQL UPDATE statements to update the
 rating of
// multiple movies that already exist in the DynamoDB table.
func (runner PartiQLRunner) UpdateMovieBatch(ctx context.Context, movies []Movie,
 ratings []float64) error {
 statementRequests := make([]types.BatchStatementRequest, len(movies))
 for index, movie := range movies {
  params, err := attributevalue.MarshalList([]interface{}{ratings[index],
 movie.Title, movie.Year})
  if err != nil {
   panic(err)
  }
  statementRequests[index] = types.BatchStatementRequest{
   Statement: aws.String(
    fmt.Sprintf("UPDATE \"%v\" SET info.rating=? WHERE title=? AND year=?",
 runner.TableName)),
   Parameters: params,
  }
 }

 _, err := runner.DynamoDbClient.BatchExecuteStatement(ctx,
 &dynamodb.BatchExecuteStatementInput{
  Statements: statementRequests,
 })
 if err != nil {
  log.Printf("Couldn't update the batch of movies. Here's why: %v\n", err)
 }
 return err
}
```

Use batches of DELETE statements to delete items.

```
// DeleteMovieBatch runs a batch of PartiQL DELETE statements to remove multiple
 movies
// from the DynamoDB table.
func (runner PartiQLRunner) DeleteMovieBatch(ctx context.Context, movies []Movie)
 error {
 statementRequests := make([]types.BatchStatementRequest, len(movies))
 for index, movie := range movies {
  params, err := attributevalue.MarshalList([]interface{}{movie.Title,
 movie.Year})
  if err != nil {
   panic(err)
  }
  statementRequests[index] = types.BatchStatementRequest{
   Statement: aws.String(
    fmt.Sprintf("DELETE FROM \"%v\" WHERE title=? AND year=?",
 runner.TableName)),
   Parameters: params,
  }
 }

 _, err := runner.DynamoDbClient.BatchExecuteStatement(ctx,
 &dynamodb.BatchExecuteStatementInput{
  Statements: statementRequests,
 })
 if err != nil {
  log.Printf("Couldn't delete the batch of movies. Here's why: %v\n", err)
 }
 return err
}
```

Define a Movie struct that is used in this example.

```
import (
 "archive/zip"
 "bytes"
 "encoding/json"
 "fmt"
 "io"
 "log"
```

```
 "net/http"

 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/attributevalue"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb/types"
)

// Movie encapsulates data about a movie. Title and Year are the composite
 primary key
// of the movie in Amazon DynamoDB. Title is the sort key, Year is the partition
 key,
// and Info is additional data.
type Movie struct {
 Title string                 `dynamodbav:"title"`
 Year  int                    `dynamodbav:"year"`
 Info  map[string]interface{} `dynamodbav:"info"`
}

// GetKey returns the composite primary key of the movie in a format that can be
// sent to DynamoDB.
func (movie Movie) GetKey() map[string]types.AttributeValue {
 title, err := attributevalue.Marshal(movie.Title)
 if err != nil {
  panic(err)
 }
 year, err := attributevalue.Marshal(movie.Year)
 if err != nil {
  panic(err)
 }
 return map[string]types.AttributeValue{"title": title, "year": year}
}

// String returns the title, year, rating, and plot of a movie, formatted for the
 example.
func (movie Movie) String() string {
 return fmt.Sprintf("%v\n\tReleased: %v\n\tRating: %v\n\tPlot: %v\n",
  movie.Title, movie.Year, movie.Info["rating"], movie.Info["plot"])
}
```

- For API details, see BatchExecuteStatement in *AWS SDK for Go API Reference*.

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Create a batch of items using PartiQL.

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";

import {
  DynamoDBDocumentClient,
  BatchExecuteStatementCommand,
} from "@aws-sdk/lib-dynamodb";

const client = new DynamoDBClient({});
const docClient = DynamoDBDocumentClient.from(client);

export const main = async () => {
  const breakfastFoods = ["Eggs", "Bacon", "Sausage"];
  const command = new BatchExecuteStatementCommand({
    Statements: breakfastFoods.map((food) => ({
      Statement: `INSERT INTO BreakfastFoods value {'Name':?}`,
      Parameters: [food],
    })),
  });

  const response = await docClient.send(command);
  console.log(response);
  return response;
};
```

Get a batch of items using PartiQL.

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";

import {
```

```
  DynamoDBDocumentClient,
  BatchExecuteStatementCommand,
} from "@aws-sdk/lib-dynamodb";

const client = new DynamoDBClient({});
const docClient = DynamoDBDocumentClient.from(client);

export const main = async () => {
  const command = new BatchExecuteStatementCommand({
    Statements: [
      {
        Statement: "SELECT * FROM PepperMeasurements WHERE Unit=?",
        Parameters: ["Teaspoons"],
        ConsistentRead: true,
      },
      {
        Statement: "SELECT * FROM PepperMeasurements WHERE Unit=?",
        Parameters: ["Grams"],
        ConsistentRead: true,
      },
    ],
  });

  const response = await docClient.send(command);
  console.log(response);
  return response;
};
```

Update a batch of items using PartiQL.

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";

import {
  DynamoDBDocumentClient,
  BatchExecuteStatementCommand,
} from "@aws-sdk/lib-dynamodb";

const client = new DynamoDBClient({});
const docClient = DynamoDBDocumentClient.from(client);

export const main = async () => {
  const eggUpdates = [
```

```
    ["duck", "fried"],
    ["chicken", "omelette"],
  ];
  const command = new BatchExecuteStatementCommand({
    Statements: eggUpdates.map((change) => ({
      Statement: "UPDATE Eggs SET Style=? where Variety=?",
      Parameters: [change[1], change[0]],
    })),
  });

  const response = await docClient.send(command);
  console.log(response);
  return response;
};
```

Delete a batch of items using PartiQL.

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";

import {
  DynamoDBDocumentClient,
  BatchExecuteStatementCommand,
} from "@aws-sdk/lib-dynamodb";

const client = new DynamoDBClient({});
const docClient = DynamoDBDocumentClient.from(client);

export const main = async () => {
  const command = new BatchExecuteStatementCommand({
    Statements: [
      {
        Statement: "DELETE FROM Flavors where Name=?",
        Parameters: ["Grape"],
      },
      {
        Statement: "DELETE FROM Flavors where Name=?",
        Parameters: ["Strawberry"],
      },
    ],
  });

  const response = await docClient.send(command);
```

```
  console.log(response);
  return response;
};
```

- For API details, see BatchExecuteStatement in *AWS SDK for JavaScript API Reference*.

PHP

SDK for PHP

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    public function getItemByPartiQLBatch(string $tableName, array $keys): Result
    {
        $statements = [];
        foreach ($keys as $key) {
            list($statement, $parameters) = $this-
>buildStatementAndParameters("SELECT", $tableName, $key['Item']);
            $statements[] = [
                'Statement' => "$statement",
                'Parameters' => $parameters,
            ];
        }

        return $this->dynamoDbClient->batchExecuteStatement([
            'Statements' => $statements,
        ]);
    }

    public function insertItemByPartiQLBatch(string $statement, array
 $parameters)
    {
        $this->dynamoDbClient->batchExecuteStatement([
            'Statements' => [
                [
                    'Statement' => "$statement",
                    'Parameters' => $parameters,
```

```
                ],
            ],
        ]);
    }

    public function updateItemByPartiQLBatch(string $statement, array
$parameters)
    {
        $this->dynamoDbClient->batchExecuteStatement([
            'Statements' => [
                [
                    'Statement' => "$statement",
                    'Parameters' => $parameters,
                ],
            ],
        ]);
    }

    public function deleteItemByPartiQLBatch(string $statement, array
$parameters)
    {
        $this->dynamoDbClient->batchExecuteStatement([
            'Statements' => [
                [
                    'Statement' => "$statement",
                    'Parameters' => $parameters,
                ],
            ],
        ]);
    }
```

- For API details, see BatchExecuteStatement in *AWS SDK for PHP API Reference*.

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
class PartiQLBatchWrapper:
    """
    Encapsulates a DynamoDB resource to run PartiQL statements.
    """

    def __init__(self, dyn_resource):
        """
        :param dyn_resource: A Boto3 DynamoDB resource.
        """
        self.dyn_resource = dyn_resource


    def run_partiql(self, statements, param_list):
        """
        Runs a PartiQL statement. A Boto3 resource is used even though
        `execute_statement` is called on the underlying `client` object because
 the
        resource transforms input and output from plain old Python objects
 (POPOs) to
        the DynamoDB format. If you create the client directly, you must do these
        transforms yourself.

        :param statements: The batch of PartiQL statements.
        :param param_list: The batch of PartiQL parameters that are associated
 with
                           each statement. This list must be in the same order as
 the
                           statements.
        :return: The responses returned from running the statements, if any.
        """
        try:
            output = self.dyn_resource.meta.client.batch_execute_statement(
                Statements=[
                    {"Statement": statement, "Parameters": params}
                    for statement, params in zip(statements, param_list)
                ]
            )
        except ClientError as err:
            if err.response["Error"]["Code"] == "ResourceNotFoundException":
                logger.error(
                    "Couldn't execute batch of PartiQL statements because the
 table "
                    "does not exist."
```

```
                )
            else:
                logger.error(
                    "Couldn't execute batch of PartiQL statements. Here's why:
 %s: %s",
                    err.response["Error"]["Code"],
                    err.response["Error"]["Message"],
                )
            raise
        else:
            return output
```

- For API details, see BatchExecuteStatement in *AWS SDK for Python (Boto3) API Reference*.

Ruby

SDK for Ruby

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Read a batch of items using PartiQL.

```
class DynamoDBPartiQLBatch
  attr_reader :dynamo_resource, :table

  def initialize(table_name)
    client = Aws::DynamoDB::Client.new(region: 'us-east-1')
    @dynamodb = Aws::DynamoDB::Resource.new(client: client)
    @table = @dynamodb.table(table_name)
  end

  # Selects a batch of items from a table using PartiQL
  #
  # @param batch_titles [Array] Collection of movie titles
  # @return [Aws::DynamoDB::Types::BatchExecuteStatementOutput]
  def batch_execute_select(batch_titles)
```

```
    request_items = batch_titles.map do |title, year|
      {
        statement: "SELECT * FROM \"#{@table.name}\" WHERE title=? and year=?",
        parameters: [title, year]
      }
    end
    @dynamodb.client.batch_execute_statement({ statements: request_items })
  end
```

Delete a batch of items using PartiQL.

```
class DynamoDBPartiQLBatch
  attr_reader :dynamo_resource, :table

  def initialize(table_name)
    client = Aws::DynamoDB::Client.new(region: 'us-east-1')
    @dynamodb = Aws::DynamoDB::Resource.new(client: client)
    @table = @dynamodb.table(table_name)
  end

  # Deletes a batch of items from a table using PartiQL
  #
  # @param batch_titles [Array] Collection of movie titles
  # @return [Aws::DynamoDB::Types::BatchExecuteStatementOutput]
  def batch_execute_write(batch_titles)
    request_items = batch_titles.map do |title, year|
      {
        statement: "DELETE FROM \"#{@table.name}\" WHERE title=? and year=?",
        parameters: [title, year]
      }
    end
    @dynamodb.client.batch_execute_statement({ statements: request_items })
  end
```

- For API details, see BatchExecuteStatement in *AWS SDK for Ruby API Reference*.

For a complete list of AWS SDK developer guides and code examples, see Using DynamoDB with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use `BatchGetItem` with an AWS SDK or CLI

The following code examples show how to use `BatchGetItem`.

.NET

SDK for .NET

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
using System;
using System.Collections.Generic;
using Amazon.DynamoDBv2;
using Amazon.DynamoDBv2.Model;

namespace LowLevelBatchGet
{
    public class LowLevelBatchGet
    {
        private static readonly string _table1Name = "Forum";
        private static readonly string _table2Name = "Thread";

        public static async void
 RetrieveMultipleItemsBatchGet(AmazonDynamoDBClient client)
        {
            var request = new BatchGetItemRequest
            {
                RequestItems = new Dictionary<string, KeysAndAttributes>()
            {
                { _table1Name,
                  new KeysAndAttributes
                  {
                      Keys = new List<Dictionary<string, AttributeValue> >()
                      {
                          new Dictionary<string, AttributeValue>()
                          {
                              { "Name", new AttributeValue {
                            S = "Amazon DynamoDB"
```

```
                        } }
                          },
                          new Dictionary<string, AttributeValue>()
                          {
                              { "Name", new AttributeValue {
                            S = "Amazon S3"
                        } }
                          }
                      }
                  }},
                {
                    _table2Name,
                    new KeysAndAttributes
                    {
                        Keys = new List<Dictionary<string, AttributeValue> >()
                        {
                            new Dictionary<string, AttributeValue>()
                            {
                                { "ForumName", new AttributeValue {
                                      S = "Amazon DynamoDB"
                                  } },
                                { "Subject", new AttributeValue {
                                      S = "DynamoDB Thread 1"
                                  } }
                            },
                            new Dictionary<string, AttributeValue>()
                            {
                                { "ForumName", new AttributeValue {
                                      S = "Amazon DynamoDB"
                                  } },
                                { "Subject", new AttributeValue {
                                      S = "DynamoDB Thread 2"
                                  } }
                            },
                            new Dictionary<string, AttributeValue>()
                            {
                                { "ForumName", new AttributeValue {
                                      S = "Amazon S3"
                                  } },
                                { "Subject", new AttributeValue {
                                      S = "S3 Thread 1"
                                  } }
                            }
                        }
```

```
                    }
                }
            }
            };

            BatchGetItemResponse response;
            do
            {
                Console.WriteLine("Making request");
                response = await client.BatchGetItemAsync(request);

                // Check the response.
                var responses = response.Responses; // Attribute list in the
response.

                foreach (var tableResponse in responses)
                {
                    var tableResults = tableResponse.Value;
                    Console.WriteLine("Items retrieved from table {0}",
tableResponse.Key);
                    foreach (var item1 in tableResults)
                    {
                        PrintItem(item1);
                    }
                }

                // Any unprocessed keys? could happen if you exceed
ProvisionedThroughput or some other error.
                Dictionary<string, KeysAndAttributes> unprocessedKeys =
response.UnprocessedKeys;
                foreach (var unprocessedTableKeys in unprocessedKeys)
                {
                    // Print table name.
                    Console.WriteLine(unprocessedTableKeys.Key);
                    // Print unprocessed primary keys.
                    foreach (var key in unprocessedTableKeys.Value.Keys)
                    {
                        PrintItem(key);
                    }
                }

                request.RequestItems = unprocessedKeys;
            } while (response.UnprocessedKeys.Count > 0);
        }
```

```
        private static void PrintItem(Dictionary<string, AttributeValue>
 attributeList)
        {
            foreach (KeyValuePair<string, AttributeValue> kvp in attributeList)
            {
                string attributeName = kvp.Key;
                AttributeValue value = kvp.Value;

                Console.WriteLine(
                    attributeName + " " +
                    (value.S == null ? "" : "S=[" + value.S + "]") +
                    (value.N == null ? "" : "N=[" + value.N + "]") +
                    (value.SS == null ? "" : "SS=[" + string.Join(",",
 value.SS.ToArray()) + "]") +
                    (value.NS == null ? "" : "NS=[" + string.Join(",",
 value.NS.ToArray()) + "]")
                    );
            }

 Console.WriteLine("************************************************");
        }

        static void Main()
        {
            var client = new AmazonDynamoDBClient();

            RetrieveMultipleItemsBatchGet(client);
        }
    }
}
```

- For API details, see BatchGetItem in *AWS SDK for .NET API Reference*.

Bash

AWS CLI with Bash script

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
##############################################################################
# function dynamodb_batch_get_item
#
# This function gets a batch of items from a DynamoDB table.
#
# Parameters:
#       -i item  -- Path to json file containing the keys of the items to get.
#
#  Returns:
#       The items as json output.
#  And:
#       0 - If successful.
#       1 - If it fails.
##############################################################################
function dynamodb_batch_get_item() {
  local item response
  local option OPTARG # Required to use getopts command in a function.

  ######################################
  # Function usage explanation
  ######################################
  function usage() {
    echo "function dynamodb_batch_get_item"
    echo "Get a batch of items from a DynamoDB table."
    echo " -i item  -- Path to json file containing the keys of the items to
 get."
    echo ""
  }

  while getopts "i:h" option; do
    case "${option}" in
      i) item="${OPTARG}" ;;
```

```
      h)
        usage
        return 0
        ;;
      \?)
        echo "Invalid parameter"
        usage
        return 1
        ;;
    esac
  done
  export OPTIND=1

  if [[ -z "$item" ]]; then
    errecho "ERROR: You must provide an item with the -i parameter."
    usage
    return 1
  fi

  response=$(aws dynamodb batch-get-item \
    --request-items file://"$item")
  local error_code=${?}

  if [[ $error_code -ne 0 ]]; then
    aws_cli_error_log $error_code
    errecho "ERROR: AWS reports batch-get-item operation failed.$response"
    return 1
  fi

  echo "$response"

  return 0
}
```

The utility functions used in this example.

```
###############################################################################
# function errecho
#
# This function outputs everything sent to it to STDERR (standard error output).
###############################################################################
function errecho() {
```

```
  printf "%s\n" "$*" 1>&2
}

###############################################################################
# function aws_cli_error_log()
#
# This function is used to log the error messages from the AWS CLI.
#
# See https://docs.aws.amazon.com/cli/latest/topic/return-codes.html#cli-aws-
help-return-codes.
#
# The function expects the following argument:
#         $1 - The error code returned by the AWS CLI.
#
#  Returns:
#          0: - Success.
#
###############################################################################
function aws_cli_error_log() {
  local err_code=$1
  errecho "Error code : $err_code"
  if [ "$err_code" == 1 ]; then
    errecho "  One or more S3 transfers failed."
  elif [ "$err_code" == 2 ]; then
    errecho "  Command line failed to parse."
  elif [ "$err_code" == 130 ]; then
    errecho "  Process received SIGINT."
  elif [ "$err_code" == 252 ]; then
    errecho "  Command syntax invalid."
  elif [ "$err_code" == 253 ]; then
    errecho "  The system environment or configuration was invalid."
  elif [ "$err_code" == 254 ]; then
    errecho "  The service returned an error."
  elif [ "$err_code" == 255 ]; then
    errecho "  255 is a catch-all error."
  fi

  return 0
}
```

- For API details, see BatchGetItem in *AWS CLI Command Reference*.

C++

SDK for C++

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
//! Batch get items from different Amazon DynamoDB tables.
/*!
  \sa batchGetItem()
  \param clientConfiguration: AWS client configuration.
  \return bool: Function succeeded.
 */
bool AwsDoc::DynamoDB::batchGetItem(
        const Aws::Client::ClientConfiguration &clientConfiguration) {
    Aws::DynamoDB::DynamoDBClient dynamoClient(clientConfiguration);

    Aws::DynamoDB::Model::BatchGetItemRequest request;

    // Table1: Forum.
    Aws::String table1Name = "Forum";
    Aws::DynamoDB::Model::KeysAndAttributes table1KeysAndAttributes;

    // Table1: Projection expression.
    table1KeysAndAttributes.SetProjectionExpression("#n, Category, Messages,
 #v");

    // Table1: Expression attribute names.
    Aws::Http::HeaderValueCollection headerValueCollection;
    headerValueCollection.emplace("#n", "Name");
    headerValueCollection.emplace("#v", "Views");
    table1KeysAndAttributes.SetExpressionAttributeNames(headerValueCollection);

    // Table1: Set key name, type, and value to search.
    std::vector<Aws::String> nameValues = {"Amazon DynamoDB", "Amazon S3"};
    for (const Aws::String &name: nameValues) {
        Aws::Map<Aws::String, Aws::DynamoDB::Model::AttributeValue> keys;
        Aws::DynamoDB::Model::AttributeValue key;
        key.SetS(name);
```

```
        keys.emplace("Name", key);
        table1KeysAndAttributes.AddKeys(keys);
    }

    Aws::Map<Aws::String, Aws::DynamoDB::Model::KeysAndAttributes> requestItems;
    requestItems.emplace(table1Name, table1KeysAndAttributes);

    // Table2: ProductCatalog.
    Aws::String table2Name = "ProductCatalog";
    Aws::DynamoDB::Model::KeysAndAttributes table2KeysAndAttributes;
    table2KeysAndAttributes.SetProjectionExpression("Title, Price, Color");

    // Table2: Set key name, type, and value to search.
    std::vector<Aws::String> idValues = {"102", "103", "201"};
    for (const Aws::String &id: idValues) {
        Aws::Map<Aws::String, Aws::DynamoDB::Model::AttributeValue> keys;
        Aws::DynamoDB::Model::AttributeValue key;
        key.SetN(id);
        keys.emplace("Id", key);
        table2KeysAndAttributes.AddKeys(keys);
    }

    requestItems.emplace(table2Name, table2KeysAndAttributes);

    bool result = true;
    do {  // Use a do loop to handle pagination.
        request.SetRequestItems(requestItems);
        const Aws::DynamoDB::Model::BatchGetItemOutcome &outcome =
dynamoClient.BatchGetItem(
                request);

        if (outcome.IsSuccess()) {
            for (const auto &responsesMapEntry:
outcome.GetResult().GetResponses()) {
                Aws::String tableName = responsesMapEntry.first;
                const Aws::Vector<Aws::Map<Aws::String,
Aws::DynamoDB::Model::AttributeValue>> &tableResults = responsesMapEntry.second;
                std::cout << "Retrieved " << tableResults.size()
                          << " responses for table '" << tableName << "'.\n"
                          << std::endl;
                if (tableName == "Forum") {

                    std::cout << "Name | Category | Message | Views" <<
std::endl;
```

```
                    for (const Aws::Map<Aws::String,
 Aws::DynamoDB::Model::AttributeValue> &item: tableResults) {
                        std::cout << item.at("Name").GetS() << " | ";
                        std::cout << item.at("Category").GetS() << " | ";
                        std::cout << (item.count("Message") == 0 ? "" : item.at(
                                "Messages").GetN()) << " | ";
                        std::cout << (item.count("Views") == 0 ? "" : item.at(
                                "Views").GetN()) << std::endl;
                    }
                }
                else {
                    std::cout << "Title | Price | Color" << std::endl;
                    for (const Aws::Map<Aws::String,
 Aws::DynamoDB::Model::AttributeValue> &item: tableResults) {
                        std::cout << item.at("Title").GetS() << " | ";
                        std::cout << (item.count("Price") == 0 ? "" : item.at(
                                "Price").GetN());
                        if (item.count("Color")) {
                            std::cout << " | ";
                            for (const
 std::shared_ptr<Aws::DynamoDB::Model::AttributeValue> &listItem: item.at(
                                    "Color").GetL())
                                std::cout << listItem->GetS() << " ";
                        }
                        std::cout << std::endl;
                    }
                }
                std::cout << std::endl;
            }

            // If necessary, repeat request for remaining items.
            requestItems = outcome.GetResult().GetUnprocessedKeys();
        }
        else {
            std::cerr << "Batch get item failed: " <<
 outcome.GetError().GetMessage()
                      << std::endl;
            result = false;
            break;
        }
    } while (!requestItems.empty());

    return result;
}
```

- For API details, see BatchGetItem in *AWS SDK for C++ API Reference*.

CLI

AWS CLI

To retrieve multiple items from a table

The following `batch-get-items` example reads multiple items from the `MusicCollection` table using a batch of three `GetItem` requests, and requests the number of read capacity units consumed by the operation. The command returns only the `AlbumTitle` attribute.

```
aws dynamodb batch-get-item \
    --request-items file://request-items.json \
    --return-consumed-capacity TOTAL
```

Contents of `request-items.json`:

```
{
    "MusicCollection": {
        "Keys": [
            {
                "Artist": {"S": "No One You Know"},
                "SongTitle": {"S": "Call Me Today"}
            },
            {
                "Artist": {"S": "Acme Band"},
                "SongTitle": {"S": "Happy Day"}
            },
            {
                "Artist": {"S": "No One You Know"},
                "SongTitle": {"S": "Scared of My Shadow"}
            }
        ],
        "ProjectionExpression":"AlbumTitle"
    }
}
```

Output:

```
{
    "Responses": {
        "MusicCollection": [
            {
                "AlbumTitle": {
                    "S": "Somewhat Famous"
                }
            },
            {
                "AlbumTitle": {
                    "S": "Blue Sky Blues"
                }
            },
            {
                "AlbumTitle": {
                    "S": "Louder Than Ever"
                }
            }
        ]
    },
    "UnprocessedKeys": {},
    "ConsumedCapacity": [
        {
            "TableName": "MusicCollection",
            "CapacityUnits": 1.5
        }
    ]
}
```

For more information, see Batch Operations in the *Amazon DynamoDB Developer Guide*.

- For API details, see BatchGetItem in *AWS CLI Command Reference*.

Java

SDK for Java 2.x

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Shows how to get batch items using the service client.

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.BatchGetItemRequest;
import software.amazon.awssdk.services.dynamodb.model.BatchGetItemResponse;
import software.amazon.awssdk.services.dynamodb.model.KeysAndAttributes;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

/**
 * Before running this Java V2 code example, set up your development environment,
 including your credentials.
 *
 * For more information, see the following documentation topic:
 *
 * https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-
started.html
 */
public class BatchReadItems {
    public static void main(String[] args){
        final String usage = """

                Usage:
                    <tableName>

                Where:
                    tableName - The Amazon DynamoDB table (for example, Music).\s
                """;

        String tableName = "Music";
```

```
        Region region = Region.US_EAST_1;
        DynamoDbClient dynamoDbClient = DynamoDbClient.builder()
            .region(region)
            .build();

        getBatchItems(dynamoDbClient, tableName);
    }

    public static void getBatchItems(DynamoDbClient dynamoDbClient, String
tableName) {
        // Define the primary key values for the items you want to retrieve.
        Map<String, AttributeValue> key1 = new HashMap<>();
        key1.put("Artist", AttributeValue.builder().s("Artist1").build());

        Map<String, AttributeValue> key2 = new HashMap<>();
        key2.put("Artist", AttributeValue.builder().s("Artist2").build());

        // Construct the batchGetItem request.
        Map<String, KeysAndAttributes> requestItems = new HashMap<>();
        requestItems.put(tableName, KeysAndAttributes.builder()
            .keys(List.of(key1, key2))
            .projectionExpression("Artist, SongTitle")
            .build());

        BatchGetItemRequest batchGetItemRequest = BatchGetItemRequest.builder()
            .requestItems(requestItems)
            .build();

        // Make the batchGetItem request.
        BatchGetItemResponse batchGetItemResponse =
dynamoDbClient.batchGetItem(batchGetItemRequest);

        // Extract and print the retrieved items.
        Map<String, List<Map<String, AttributeValue>>> responses =
batchGetItemResponse.responses();
        if (responses.containsKey(tableName)) {
            List<Map<String, AttributeValue>> musicItems =
responses.get(tableName);
            for (Map<String, AttributeValue> item : musicItems) {
                System.out.println("Artist: " + item.get("Artist").s() +
                    ", SongTitle: " + item.get("SongTitle").s());
            }
        } else {
            System.out.println("No items retrieved.");
```

```
        }
    }
}
```

Shows how to get batch items using the service client and a paginator.

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.BatchGetItemRequest;
import software.amazon.awssdk.services.dynamodb.model.KeysAndAttributes;
import java.util.Collections;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

public class BatchGetItemsPaginator {

    public static void main(String[] args){
        final String usage = """

                Usage:
                    <tableName>

                Where:
                    tableName - The Amazon DynamoDB table (for example, Music).\s
                """;

        String tableName = "Music";
        Region region = Region.US_EAST_1;
        DynamoDbClient dynamoDbClient = DynamoDbClient.builder()
            .region(region)
            .build();

        getBatchItemsPaginator(dynamoDbClient, tableName) ;
    }

    public static void getBatchItemsPaginator(DynamoDbClient dynamoDbClient,
 String tableName) {
        // Define the primary key values for the items you want to retrieve.
        Map<String, AttributeValue> key1 = new HashMap<>();
        key1.put("Artist", AttributeValue.builder().s("Artist1").build());
```

```
        Map<String, AttributeValue> key2 = new HashMap<>();
        key2.put("Artist", AttributeValue.builder().s("Artist2").build());

        // Construct the batchGetItem request.
        Map<String, KeysAndAttributes> requestItems = new HashMap<>();
        requestItems.put(tableName, KeysAndAttributes.builder()
            .keys(List.of(key1, key2))
            .projectionExpression("Artist, SongTitle")
            .build());

        BatchGetItemRequest batchGetItemRequest = BatchGetItemRequest.builder()
            .requestItems(requestItems)
            .build();

        // Use batchGetItemPaginator for paginated requests.
        dynamoDbClient.batchGetItemPaginator(batchGetItemRequest).stream()
            .flatMap(response -> response.responses().getOrDefault(tableName,
 Collections.emptyList()).stream())
            .forEach(item -> {
                System.out.println("Artist: " + item.get("Artist").s() +
                    ", SongTitle: " + item.get("SongTitle").s());
            });
    }
}
```

- For API details, see BatchGetItem in *AWS SDK for Java 2.x API Reference*.

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

This example uses the document client to simplify working with items in DynamoDB. For API details see BatchGet.

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
import { BatchGetCommand, DynamoDBDocumentClient } from "@aws-sdk/lib-dynamodb";

const client = new DynamoDBClient({});
const docClient = DynamoDBDocumentClient.from(client);

export const main = async () => {
  const command = new BatchGetCommand({
    // Each key in this object is the name of a table. This example refers
    // to a Books table.
    RequestItems: {
      Books: {
        // Each entry in Keys is an object that specifies a primary key.
        Keys: [
          {
            Title: "How to AWS",
          },
          {
            Title: "DynamoDB for DBAs",
          },
        ],
        // Only return the "Title" and "PageCount" attributes.
        ProjectionExpression: "Title, PageCount",
      },
    },
  });

  const response = await docClient.send(command);
  console.log(response.Responses.Books);
  return response;
};
```

- For more information, see AWS SDK for JavaScript Developer Guide.
- For API details, see BatchGetItem in *AWS SDK for JavaScript API Reference*.

SDK for JavaScript (v2)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create DynamoDB service object
var ddb = new AWS.DynamoDB({ apiVersion: "2012-08-10" });

var params = {
  RequestItems: {
    TABLE_NAME: {
      Keys: [
        { KEY_NAME: { N: "KEY_VALUE_1" } },
        { KEY_NAME: { N: "KEY_VALUE_2" } },
        { KEY_NAME: { N: "KEY_VALUE_3" } },
      ],
      ProjectionExpression: "KEY_NAME, ATTRIBUTE",
    },
  },
};

ddb.batchGetItem(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    data.Responses.TABLE_NAME.forEach(function (element, index, array) {
      console.log(element);
    });
  }
});
```

- For more information, see AWS SDK for JavaScript Developer Guide.
- For API details, see BatchGetItem in *AWS SDK for JavaScript API Reference*.

PowerShell

Tools for PowerShell V4

Example 1: Gets the item with the SongTitle "Somewhere Down The Road" from the DynamoDB tables 'Music' and 'Songs'.

```
$key = @{
    SongTitle = 'Somewhere Down The Road'
    Artist = 'No One You Know'
} | ConvertTo-DDBItem

$keysAndAttributes = New-Object Amazon.DynamoDBv2.Model.KeysAndAttributes
$list = New-Object
 'System.Collections.Generic.List[System.Collections.Generic.Dictionary[String,
 Amazon.DynamoDBv2.Model.AttributeValue]]'
$list.Add($key)
$keysAndAttributes.Keys = $list

$requestItem = @{
    'Music' = [Amazon.DynamoDBv2.Model.KeysAndAttributes]$keysAndAttributes
    'Songs' = [Amazon.DynamoDBv2.Model.KeysAndAttributes]$keysAndAttributes
}

$batchItems = Get-DDBBatchItem -RequestItem $requestItem
$batchItems.GetEnumerator() | ForEach-Object {$PSItem.Value} | ConvertFrom-
DDBItem
```

Output:

```
Name                           Value
----                           -----
Artist                         No One You Know
SongTitle                      Somewhere Down The Road
AlbumTitle                     Somewhat Famous
CriticRating                   10
Genre                          Country
Price                          1.94
Artist                         No One You Know
SongTitle                      Somewhere Down The Road
AlbumTitle                     Somewhat Famous
CriticRating                   10
Genre                          Country
Price                          1.94
```

- For API details, see BatchGetItem in *AWS Tools for PowerShell Cmdlet Reference (V4)*.

Tools for PowerShell V5

Example 1: Gets the item with the SongTitle "Somewhere Down The Road" from the DynamoDB tables 'Music' and 'Songs'.

```
$key = @{
    SongTitle = 'Somewhere Down The Road'
    Artist = 'No One You Know'
} | ConvertTo-DDBItem

$keysAndAttributes = New-Object Amazon.DynamoDBv2.Model.KeysAndAttributes
$list = New-Object
 'System.Collections.Generic.List[System.Collections.Generic.Dictionary[String,
 Amazon.DynamoDBv2.Model.AttributeValue]]'
$list.Add($key)
$keysAndAttributes.Keys = $list

$requestItem = @{
    'Music' = [Amazon.DynamoDBv2.Model.KeysAndAttributes]$keysAndAttributes
    'Songs' = [Amazon.DynamoDBv2.Model.KeysAndAttributes]$keysAndAttributes
}

$batchItems = Get-DDBBatchItem -RequestItem $requestItem
$batchItems.GetEnumerator() | ForEach-Object {$PSItem.Value} | ConvertFrom-
DDBItem
```

Output:

```
Name                           Value
----                           -----
Artist                         No One You Know
SongTitle                      Somewhere Down The Road
AlbumTitle                     Somewhat Famous
CriticRating                   10
Genre                          Country
Price                          1.94
Artist                         No One You Know
SongTitle                      Somewhere Down The Road
AlbumTitle                     Somewhat Famous
CriticRating                   10
Genre                          Country
Price                          1.94
```

- For API details, see BatchGetItem in *AWS Tools for PowerShell Cmdlet Reference (V5)*.

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import decimal
import json
import logging
import os
import pprint
import time
import boto3
from botocore.exceptions import ClientError

logger = logging.getLogger(__name__)
dynamodb = boto3.resource("dynamodb")

MAX_GET_SIZE = 100  # Amazon DynamoDB rejects a get batch larger than 100 items.


def do_batch_get(batch_keys):
    """
    Gets a batch of items from Amazon DynamoDB. Batches can contain keys from
    more than one table.

    When Amazon DynamoDB cannot process all items in a batch, a set of
 unprocessed
    keys is returned. This function uses an exponential backoff algorithm to
 retry
    getting the unprocessed keys until all are retrieved or the specified
    number of tries is reached.

    :param batch_keys: The set of keys to retrieve. A batch can contain at most
 100
                       keys. Otherwise, Amazon DynamoDB returns an error.
```

```
    :return: The dictionary of retrieved items grouped under their respective
             table names.
    """
    tries = 0
    max_tries = 5
    sleepy_time = 1  # Start with 1 second of sleep, then exponentially increase.
    retrieved = {key: [] for key in batch_keys}
    while tries < max_tries:
        response = dynamodb.batch_get_item(RequestItems=batch_keys)
        # Collect any retrieved items and retry unprocessed keys.
        for key in response.get("Responses", []):
            retrieved[key] += response["Responses"][key]
        unprocessed = response["UnprocessedKeys"]
        if len(unprocessed) > 0:
            batch_keys = unprocessed
            unprocessed_count = sum(
                [len(batch_key["Keys"]) for batch_key in batch_keys.values()]
            )
            logger.info(
                "%s unprocessed keys returned. Sleep, then retry.",
unprocessed_count
            )
            tries += 1
            if tries < max_tries:
                logger.info("Sleeping for %s seconds.", sleepy_time)
                time.sleep(sleepy_time)
                sleepy_time = min(sleepy_time * 2, 32)
        else:
            break

    return retrieved
```

- For API details, see BatchGetItem in *AWS SDK for Python (Boto3) API Reference*.

Swift

SDK for Swift

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import AWSDynamoDB


    /// Gets an array of `Movie` objects describing all the movies in the
    /// specified list. Any movies that aren't found in the list have no
    /// corresponding entry in the resulting array.
    ///
    /// - Parameters
    ///     - keys: An array of tuples, each of which specifies the title and
    ///       release year of a movie to fetch from the table.
    ///
    /// - Returns:
    ///     - An array of `Movie` objects describing each match found in the
    ///     table.
    ///
    /// - Throws:
    ///     - `MovieError.ClientUninitialized` if the DynamoDB client has not
    ///     been initialized.
    ///     - DynamoDB errors are thrown without change.
    func batchGet(keys: [(title: String, year: Int)]) async throws -> [Movie] {
        do {
            guard let client = self.ddbClient else {
                throw MovieError.ClientUninitialized
            }

            var movieList: [Movie] = []
            var keyItems: [[Swift.String: DynamoDBClientTypes.AttributeValue]] =
 []

            // Convert the list of keys into the form used by DynamoDB.

            for key in keys {
```

```
                let item: [Swift.String: DynamoDBClientTypes.AttributeValue] = [
                    "title": .s(key.title),
                    "year": .n(String(key.year))
                ]
                keyItems.append(item)
            }

            // Create the input record for `batchGetItem()`. The list of
requested
            // items is in the `requestItems` property. This array contains one
            // entry for each table from which items are to be fetched. In this
            // example, there's only one table containing the movie data.
            //
            // If we wanted this program to also support searching for matches
            // in a table of book data, we could add a second `requestItem`
            // mapping the name of the book table to the list of items we want to
            // find in it.
            let input = BatchGetItemInput(
                requestItems: [
                    self.tableName: .init(
                        consistentRead: true,
                        keys: keyItems
                    )
                ]
            )

            // Fetch the matching movies from the table.

            let output = try await client.batchGetItem(input: input)

            // Get the set of responses. If there aren't any, return the empty
            // movie list.

            guard let responses = output.responses else {
                return movieList
            }

            // Get the list of matching items for the table with the name
            // `tableName`.

            guard let responseList = responses[self.tableName] else {
                return movieList
            }
```

```
            // Create `Movie` items for each of the matching movies in the table
            // and add them to the `MovieList` array.

            for response in responseList {
                try movieList.append(Movie(withItem: response))
            }

            return movieList
        } catch {
            print("ERROR: batchGet", dump(error))
            throw error
        }
    }
```

- For API details, see BatchGetItem in *AWS SDK for Swift API reference*.

For a complete list of AWS SDK developer guides and code examples, see Using DynamoDB with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use `BatchWriteItem` with an AWS SDK or CLI

The following code examples show how to use `BatchWriteItem`.

Action examples are code excerpts from larger programs and must be run in context. You can see this action in context in the following code example:

- Learn the basics

.NET

SDK for .NET (v4)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Writes a batch of items to the movie table.

```
/// <summary>
/// Loads the contents of a JSON file into a list of movies to be
/// added to the DynamoDB table.
/// </summary>
/// <param name="movieFileName">The name of the JSON file.</param>
/// <returns>A generic list of movie objects.</returns>
public List<Movie> ImportMovies(string movieFileName)
{
    var moviesList = new List<Movie>();
    if (!File.Exists(movieFileName))
    {
        return moviesList;
    }

    using var sr = new StreamReader(movieFileName);
    string json = sr.ReadToEnd();
    var allMovies = JsonSerializer.Deserialize<List<Movie>>(
        json,
        new JsonSerializerOptions
        {
            PropertyNameCaseInsensitive = true
        });

    // Now return the first 250 entries.
    if (allMovies != null && allMovies.Any())
    {
        moviesList = allMovies.GetRange(0, 250);
    }
    return moviesList;
}

/// <summary>
/// Writes 250 items to the movie table.
/// </summary>
/// <param name="movieFileName">A string containing the full path to
/// the JSON file containing movie data.</param>
/// <param name="tableName">The name of the table to write items to.</param>
/// <returns>A long integer value representing the number of movies
/// imported from the JSON file.</returns>
public async Task<long> BatchWriteItemsAsync(
    string movieFileName, string tableName)
{
```

```
        try
        {
            var movies = ImportMovies(movieFileName);
            if (!movies.Any())
            {
                Console.WriteLine("Couldn't find the JSON file with movie
data.");
                return 0;
            }

            var context = new DynamoDBContextBuilder()
                // Optional call to provide a specific instance of
IAmazonDynamoDB
                .WithDynamoDBClient(() => _amazonDynamoDB)
                .Build();

            var movieBatch = context.CreateBatchWrite<Movie>(
                new BatchWriteConfig()
                {
                    OverrideTableName = tableName
                });
            movieBatch.AddPutItems(movies);

            Console.WriteLine("Adding imported movies to the table.");
            await movieBatch.ExecuteAsync();

            return movies.Count;
        }
        catch (ResourceNotFoundException ex)
        {
            Console.WriteLine($"Table was not found during batch write operation.
{ex.Message}");
            throw;
        }
        catch (AmazonDynamoDBException ex)
        {
            Console.WriteLine($"An Amazon DynamoDB error occurred during batch
write operation. {ex.Message}");
            throw;
        }
        catch (Exception ex)
        {
            Console.WriteLine($"An error occurred during batch write operation.
{ex.Message}");
```

```
            throw;
        }
    }
```

- For API details, see BatchWriteItem in *AWS SDK for .NET API Reference*.

Bash

AWS CLI with Bash script

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
###############################################################################
# function dynamodb_batch_write_item
#
# This function writes a batch of items into a DynamoDB table.
#
# Parameters:
#       -i item  -- Path to json file containing the items to write.
#
#  Returns:
#       0 - If successful.
#       1 - If it fails.
###############################################################################
function dynamodb_batch_write_item() {
  local item response
  local option OPTARG # Required to use getopts command in a function.

  ######################################
  # Function usage explanation
  ######################################
  function usage() {
    echo "function dynamodb_batch_write_item"
    echo "Write a batch of items into a DynamoDB table."
    echo " -i item  -- Path to json file containing the items to write."
    echo ""
```

```
  }
  while getopts "i:h" option; do
    case "${option}" in
      i) item="${OPTARG}" ;;
      h)
        usage
        return 0
        ;;
      \?)
        echo "Invalid parameter"
        usage
        return 1
        ;;
    esac
  done
  export OPTIND=1

  if [[ -z "$item" ]]; then
    errecho "ERROR: You must provide an item with the -i parameter."
    usage
    return 1
  fi

  iecho "Parameters:\n"
  iecho "    table_name:   $table_name"
  iecho "    item:   $item"
  iecho ""

  response=$(aws dynamodb batch-write-item \
    --request-items file://"$item")

  local error_code=${?}

  if [[ $error_code -ne 0 ]]; then
    aws_cli_error_log $error_code
    errecho "ERROR: AWS reports batch-write-item operation failed.$response"
    return 1
  fi

  return 0
}
```

The utility functions used in this example.

```
##############################################################################
# function iecho
#
# This function enables the script to display the specified text only if
# the global variable $VERBOSE is set to true.
##############################################################################
function iecho() {
  if [[ $VERBOSE == true ]]; then
    echo "$@"
  fi
}

##############################################################################
# function errecho
#
# This function outputs everything sent to it to STDERR (standard error output).
##############################################################################
function errecho() {
  printf "%s\n" "$*" 1>&2
}

##############################################################################
# function aws_cli_error_log()
#
# This function is used to log the error messages from the AWS CLI.
#
# See https://docs.aws.amazon.com/cli/latest/topic/return-codes.html#cli-aws-
help-return-codes.
#
# The function expects the following argument:
#         $1 - The error code returned by the AWS CLI.
#
#  Returns:
#          0: - Success.
#
##############################################################################
function aws_cli_error_log() {
  local err_code=$1
  errecho "Error code : $err_code"
  if [ "$err_code" == 1 ]; then
    errecho "  One or more S3 transfers failed."
  elif [ "$err_code" == 2 ]; then
```

```
    errecho "  Command line failed to parse."
  elif [ "$err_code" == 130 ]; then
    errecho "  Process received SIGINT."
  elif [ "$err_code" == 252 ]; then
    errecho "  Command syntax invalid."
  elif [ "$err_code" == 253 ]; then
    errecho "  The system environment or configuration was invalid."
  elif [ "$err_code" == 254 ]; then
    errecho "  The service returned an error."
  elif [ "$err_code" == 255 ]; then
    errecho "  255 is a catch-all error."
  fi

  return 0
}
```

- For API details, see BatchWriteItem in *AWS CLI Command Reference*.

C++

SDK for C++

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
//! Batch write items from a JSON file.
/*!
  \sa batchWriteItem()
  \param jsonFilePath: JSON file path.
  \param clientConfiguration: AWS client configuration.
  \return bool: Function succeeded.
 */

/*
 * The input for this routine is a JSON file that you can download from the
 following URL:
 * https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/
SampleData.html.
```

```
 *
 * The JSON data uses the BatchWriteItem API request syntax. The JSON strings are
 * converted to AttributeValue objects. These AttributeValue objects will then
 generate
 * JSON strings when constructing the BatchWriteItem request, essentially
 outputting
 * their input.
 *
 * This is perhaps an artificial example, but it demonstrates the APIs.
 */

bool AwsDoc::DynamoDB::batchWriteItem(const Aws::String &jsonFilePath,
                                      const Aws::Client::ClientConfiguration
 &clientConfiguration) {
    std::ifstream fileStream(jsonFilePath);

    if (!fileStream) {
        std::cerr << "Error: could not open file '" << jsonFilePath << "'."
                  << std::endl;
    }

    std::stringstream stringStream;
    stringStream << fileStream.rdbuf();
    Aws::Utils::Json::JsonValue jsonValue(stringStream);

    Aws::DynamoDB::Model::BatchWriteItemRequest batchWriteItemRequest;
    Aws::Map<Aws::String, Aws::Utils::Json::JsonView> level1Map =
 jsonValue.View().GetAllObjects();
    for (const auto &level1Entry: level1Map) {
        const Aws::Utils::Json::JsonView &entriesView = level1Entry.second;
        const Aws::String &tableName = level1Entry.first;
        // The JSON entries at this level are as follows:
        //  key - table name
        //  value - list of request objects
        if (!entriesView.IsListType()) {
            std::cerr << "Error: JSON file entry '"
                      << tableName << "' is not a list." << std::endl;
            continue;
        }

        Aws::Utils::Array<Aws::Utils::Json::JsonView> entries =
 entriesView.AsArray();

        Aws::Vector<Aws::DynamoDB::Model::WriteRequest> writeRequests;
```

```
        if (AwsDoc::DynamoDB::addWriteRequests(tableName, entries,
                                               writeRequests)) {
            batchWriteItemRequest.AddRequestItems(tableName, writeRequests);
        }
    }

    Aws::DynamoDB::DynamoDBClient dynamoClient(clientConfiguration);

    Aws::DynamoDB::Model::BatchWriteItemOutcome outcome =
 dynamoClient.BatchWriteItem(
            batchWriteItemRequest);

    if (outcome.IsSuccess()) {
        std::cout << "DynamoDB::BatchWriteItem was successful." << std::endl;
    }
    else {
        std::cerr << "Error with DynamoDB::BatchWriteItem. "
                  << outcome.GetError().GetMessage()
                  << std::endl;
        return false;
    }

    return outcome.IsSuccess();
}

//! Convert requests in JSON format to a vector of WriteRequest objects.
/*!
  \sa addWriteRequests()
  \param tableName: Name of the table for the write operations.
  \param requestsJson: Request data in JSON format.
  \param writeRequests: Vector to receive the WriteRequest objects.
  \return bool: Function succeeded.
 */
bool AwsDoc::DynamoDB::addWriteRequests(const Aws::String &tableName,
                                        const
 Aws::Utils::Array<Aws::Utils::Json::JsonView> &requestsJson,

 Aws::Vector<Aws::DynamoDB::Model::WriteRequest> &writeRequests) {
    for (size_t i = 0; i < requestsJson.GetLength(); ++i) {
        const Aws::Utils::Json::JsonView &requestsEntry = requestsJson[i];
        if (!requestsEntry.IsObject()) {
            std::cerr << "Error: incorrect requestsEntry type "
                      << requestsEntry.WriteReadable() << std::endl;
            return false;
```

```
        }

        Aws::Map<Aws::String, Aws::Utils::Json::JsonView> requestsMap =
 requestsEntry.GetAllObjects();

        for (const auto &request: requestsMap) {
            const Aws::String &requestType = request.first;
            const Aws::Utils::Json::JsonView &requestJsonView = request.second;

            if (requestType == "PutRequest") {
                if (!requestJsonView.ValueExists("Item")) {
                    std::cerr << "Error: item key missing for requests "
                              << requestJsonView.WriteReadable() << std::endl;
                    return false;
                }
                Aws::Map<Aws::String, Aws::DynamoDB::Model::AttributeValue>
 attributes;
                if (!getAttributeObjectsMap(requestJsonView.GetObject("Item"),
                                            attributes)) {
                    std::cerr << "Error getting attributes "
                              << requestJsonView.WriteReadable() << std::endl;
                    return false;
                }

                Aws::DynamoDB::Model::PutRequest putRequest;
                putRequest.SetItem(attributes);
                writeRequests.push_back(
                        Aws::DynamoDB::Model::WriteRequest().WithPutRequest(
                                putRequest));
            }
            else {
                std::cerr << "Error: unimplemented request type '" << requestType
                          << "'." << std::endl;
            }
        }
    }

    return true;
}

//! Generate a map of AttributeValue objects from JSON records.
/*!
  \sa getAttributeObjectsMap()
  \param jsonView: JSONView of attribute records.
```

```
  \param writeRequests: Map to receive the AttributeValue objects.
  \return bool: Function succeeded.
 */
bool
AwsDoc::DynamoDB::getAttributeObjectsMap(const Aws::Utils::Json::JsonView
 &jsonView,
                                         Aws::Map<Aws::String,
 Aws::DynamoDB::Model::AttributeValue> &attributes) {
    Aws::Map<Aws::String, Aws::Utils::Json::JsonView> objectsMap =
 jsonView.GetAllObjects();
    for (const auto &entry: objectsMap) {
        const Aws::String &attributeKey = entry.first;
        const Aws::Utils::Json::JsonView &attributeJsonView = entry.second;

        if (!attributeJsonView.IsObject()) {
            std::cerr << "Error: attribute not an object "
                      << attributeJsonView.WriteReadable() << std::endl;
            return false;
        }

        attributes.emplace(attributeKey,

 Aws::DynamoDB::Model::AttributeValue(attributeJsonView));
    }

    return true;
}
```

- For API details, see BatchWriteItem in *AWS SDK for C++ API Reference*.

CLI

AWS CLI

To add multiple items to a table

The following `batch-write-item` example adds three new items to the `MusicCollection` table using a batch of three `PutItem` requests. It also requests information about the number of write capacity units consumed by the operation and any item collections modified by the operation.

```
aws dynamodb batch-write-item \
    --request-items file://request-items.json \
    --return-consumed-capacity INDEXES \
    --return-item-collection-metrics SIZE
```

Contents of `request-items.json`:

```
{
    "MusicCollection": [
        {
            "PutRequest": {
                "Item": {
                    "Artist": {"S": "No One You Know"},
                    "SongTitle": {"S": "Call Me Today"},
                    "AlbumTitle": {"S": "Somewhat Famous"}
                }
            }
        },
        {
            "PutRequest": {
                "Item": {
                    "Artist": {"S": "Acme Band"},
                    "SongTitle": {"S": "Happy Day"},
                    "AlbumTitle": {"S": "Songs About Life"}
                }
            }
        },
        {
            "PutRequest": {
                "Item": {
                    "Artist": {"S": "No One You Know"},
                    "SongTitle": {"S": "Scared of My Shadow"},
                    "AlbumTitle": {"S": "Blue Sky Blues"}
                }
            }
        }
    ]
}
```

Output:

```
{
```

```
    "UnprocessedItems": {},
    "ItemCollectionMetrics": {
        "MusicCollection": [
            {
                "ItemCollectionKey": {
                    "Artist": {
                        "S": "No One You Know"
                    }
                },
                "SizeEstimateRangeGB": [
                    0.0,
                    1.0
                ]
            },
            {
                "ItemCollectionKey": {
                    "Artist": {
                        "S": "Acme Band"
                    }
                },
                "SizeEstimateRangeGB": [
                    0.0,
                    1.0
                ]
            }
        ]
    },
    "ConsumedCapacity": [
        {
            "TableName": "MusicCollection",
            "CapacityUnits": 6.0,
            "Table": {
                "CapacityUnits": 3.0
            },
            "LocalSecondaryIndexes": {
                "AlbumTitleIndex": {
                    "CapacityUnits": 3.0
                }
            }
        }
    ]
}
```

For more information, see Batch Operations in the *Amazon DynamoDB Developer Guide*.

- For API details, see BatchWriteItem in *AWS CLI Command Reference*.

Go

SDK for Go V2

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import (
 "context"
 "errors"
 "log"
 "time"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/attributevalue"
 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/expression"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb/types"
)

// TableBasics encapsulates the Amazon DynamoDB service actions used in the
 examples.
// It contains a DynamoDB service client that is used to act on the specified
 table.
type TableBasics struct {
 DynamoDbClient *dynamodb.Client
 TableName      string
}



// AddMovieBatch adds a slice of movies to the DynamoDB table. The function sends
// batches of 25 movies to DynamoDB until all movies are added or it reaches the
// specified maximum.
```

```
func (basics TableBasics) AddMovieBatch(ctx context.Context, movies []Movie,
 maxMovies int) (int, error) {
 var err error
 var item map[string]types.AttributeValue
 written := 0
 batchSize := 25 // DynamoDB allows a maximum batch size of 25 items.
 start := 0
 end := start + batchSize
 for start < maxMovies && start < len(movies) {
  var writeReqs []types.WriteRequest
  if end > len(movies) {
   end = len(movies)
  }
  for _, movie := range movies[start:end] {
   item, err = attributevalue.MarshalMap(movie)
   if err != nil {
    log.Printf("Couldn't marshal movie %v for batch writing. Here's why: %v\n",
 movie.Title, err)
   } else {
    writeReqs = append(
     writeReqs,
     types.WriteRequest{PutRequest: &types.PutRequest{Item: item}},
    )
   }
  }
  _, err = basics.DynamoDbClient.BatchWriteItem(ctx,
 &dynamodb.BatchWriteItemInput{
   RequestItems: map[string][]types.WriteRequest{basics.TableName: writeReqs}})
  if err != nil {
   log.Printf("Couldn't add a batch of movies to %v. Here's why: %v\n",
 basics.TableName, err)
  } else {
   written += len(writeReqs)
  }
  start = end
  end += batchSize
 }

 return written, err
}
```

Define a Movie struct that is used in this example.

```
import (
 "archive/zip"
 "bytes"
 "encoding/json"
 "fmt"
 "io"
 "log"
 "net/http"

 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/attributevalue"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb/types"
)

// Movie encapsulates data about a movie. Title and Year are the composite
 primary key
// of the movie in Amazon DynamoDB. Title is the sort key, Year is the partition
 key,
// and Info is additional data.
type Movie struct {
 Title string                 `dynamodbav:"title"`
 Year  int                    `dynamodbav:"year"`
 Info  map[string]interface{} `dynamodbav:"info"`
}

// GetKey returns the composite primary key of the movie in a format that can be
// sent to DynamoDB.
func (movie Movie) GetKey() map[string]types.AttributeValue {
 title, err := attributevalue.Marshal(movie.Title)
 if err != nil {
  panic(err)
 }
 year, err := attributevalue.Marshal(movie.Year)
 if err != nil {
  panic(err)
 }
 return map[string]types.AttributeValue{"title": title, "year": year}
}

// String returns the title, year, rating, and plot of a movie, formatted for the
 example.
func (movie Movie) String() string {
```

```
 return fmt.Sprintf("%v\n\tReleased: %v\n\tRating: %v\n\tPlot: %v\n",
  movie.Title, movie.Year, movie.Info["rating"], movie.Info["plot"])
}
```

- For API details, see BatchWriteItem in *AWS SDK for Go API Reference*.

Java

SDK for Java 2.x

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Inserts many items into a table by using the service client.

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.BatchWriteItemRequest;
import software.amazon.awssdk.services.dynamodb.model.BatchWriteItemResponse;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.PutRequest;
import software.amazon.awssdk.services.dynamodb.model.WriteRequest;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

/**
 * Before running this Java V2 code example, set up your development environment,
 including your credentials.
 *
 * For more information, see the following documentation topic:
 *
 * https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-
started.html
 */
```

```
public class BatchWriteItems {
    public static void main(String[] args){
        final String usage = """

                Usage:
                    <tableName>

                Where:
                    tableName - The Amazon DynamoDB table (for example, Music).\s
                """;

        String tableName = "Music";
        Region region = Region.US_EAST_1;
        DynamoDbClient dynamoDbClient = DynamoDbClient.builder()
            .region(region)
            .build();

        addBatchItems(dynamoDbClient, tableName);
    }

    public static void addBatchItems(DynamoDbClient dynamoDbClient, String
 tableName) {
        // Specify the updates you want to perform.
        List<WriteRequest> writeRequests = new ArrayList<>();

        // Set item 1.
        Map<String, AttributeValue> item1Attributes = new HashMap<>();
        item1Attributes.put("Artist",
 AttributeValue.builder().s("Artist1").build());
        item1Attributes.put("Rating", AttributeValue.builder().s("5").build());
        item1Attributes.put("Comments", AttributeValue.builder().s("Great
 song!").build());
        item1Attributes.put("SongTitle",
 AttributeValue.builder().s("SongTitle1").build());

 writeRequests.add(WriteRequest.builder().putRequest(PutRequest.builder().item(item1Attri

        // Set item 2.
        Map<String, AttributeValue> item2Attributes = new HashMap<>();
        item2Attributes.put("Artist",
 AttributeValue.builder().s("Artist2").build());
        item2Attributes.put("Rating", AttributeValue.builder().s("4").build());
        item2Attributes.put("Comments", AttributeValue.builder().s("Nice
 melody.").build());
```

```
        item2Attributes.put("SongTitle",
 AttributeValue.builder().s("SongTitle2").build());

 writeRequests.add(WriteRequest.builder().putRequest(PutRequest.builder().item(item2Attri

        try {
            // Create the BatchWriteItemRequest.
            BatchWriteItemRequest batchWriteItemRequest =
 BatchWriteItemRequest.builder()
                .requestItems(Map.of(tableName, writeRequests))
                .build();

            // Execute the BatchWriteItem operation.
            BatchWriteItemResponse batchWriteItemResponse =
 dynamoDbClient.batchWriteItem(batchWriteItemRequest);

            // Process the response.
            System.out.println("Batch write successful: " +
 batchWriteItemResponse);

        } catch (DynamoDbException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }
}
```

Inserts many items into a table by using the enhanced client.

```
import com.example.dynamodb.Customer;
import com.example.dynamodb.Music;
import software.amazon.awssdk.enhanced.dynamodb.DynamoDbEnhancedClient;
import software.amazon.awssdk.enhanced.dynamodb.DynamoDbTable;
import software.amazon.awssdk.enhanced.dynamodb.Key;
import software.amazon.awssdk.enhanced.dynamodb.TableSchema;
import
 software.amazon.awssdk.enhanced.dynamodb.model.BatchWriteItemEnhancedRequest;
import software.amazon.awssdk.enhanced.dynamodb.model.WriteBatch;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import java.time.Instant;
```

```
import java.time.LocalDate;
import java.time.LocalDateTime;
import java.time.ZoneOffset;

/*
 * Before running this code example, create an Amazon DynamoDB table named
 Customer with these columns:
 *   - id - the id of the record that is the key
 *   - custName - the customer name
 *   - email - the email value
 *   - registrationDate - an instant value when the item was added to the table
 *
 * Also, ensure that you have set up your development environment, including your
 credentials.
 *
 * For information, see this documentation topic:
 *
 * https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-
started.html
 */
public class EnhancedBatchWriteItems {
        public static void main(String[] args) {
                Region region = Region.US_EAST_1;
                DynamoDbClient ddb = DynamoDbClient.builder()
                                .region(region)
                                .build();
                DynamoDbEnhancedClient enhancedClient =
 DynamoDbEnhancedClient.builder()
                                .dynamoDbClient(ddb)
                                .build();
                putBatchRecords(enhancedClient);
                ddb.close();
        }

        public static void putBatchRecords(DynamoDbEnhancedClient enhancedClient)
 {
                try {
                        DynamoDbTable<Customer> customerMappedTable =
 enhancedClient.table("Customer",
                                        TableSchema.fromBean(Customer.class));
                        DynamoDbTable<Music> musicMappedTable =
 enhancedClient.table("Music",
                                        TableSchema.fromBean(Music.class));
                        LocalDate localDate = LocalDate.parse("2020-04-07");
```

```
                        LocalDateTime localDateTime = localDate.atStartOfDay();
                        Instant instant =
localDateTime.toInstant(ZoneOffset.UTC);

                        Customer record2 = new Customer();
                        record2.setCustName("Fred Pink");
                        record2.setId("id110");
                        record2.setEmail("fredp@noserver.com");
                        record2.setRegistrationDate(instant);

                        Customer record3 = new Customer();
                        record3.setCustName("Susan Pink");
                        record3.setId("id120");
                        record3.setEmail("spink@noserver.com");
                        record3.setRegistrationDate(instant);

                        Customer record4 = new Customer();
                        record4.setCustName("Jerry orange");
                        record4.setId("id101");
                        record4.setEmail("jorange@noserver.com");
                        record4.setRegistrationDate(instant);

                        BatchWriteItemEnhancedRequest
batchWriteItemEnhancedRequest = BatchWriteItemEnhancedRequest
                                        .builder()
                                        .writeBatches(

WriteBatch.builder(Customer.class) // add items to the Customer

         // table

.mappedTableResource(customerMappedTable)

.addPutItem(builder -> builder.item(record2))

.addPutItem(builder -> builder.item(record3))

.addPutItem(builder -> builder.item(record4))
                                                                        .build(),

WriteBatch.builder(Music.class) // delete an item from the Music

      // table
```

```
 .mappedTableResource(musicMappedTable)

 .addDeleteItem(builder -> builder.key(

       Key.builder().partitionValue(

                        "Famous Band")

                        .build()))
                                                                    .build())
                                    .build();

                         // Add three items to the Customer table and delete one
 item from the Music
                         // table.

 enhancedClient.batchWriteItem(batchWriteItemEnhancedRequest);
                         System.out.println("done");

                } catch (DynamoDbException e) {
                        System.err.println(e.getMessage());
                        System.exit(1);
                }
        }
}
```

- For API details, see BatchWriteItem in *AWS SDK for Java 2.x API Reference*.

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

This example uses the document client to simplify working with items in DynamoDB. For API details see BatchWrite.

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
import {
  BatchWriteCommand,
  DynamoDBDocumentClient,
} from "@aws-sdk/lib-dynamodb";
import { readFileSync } from "node:fs";

// These modules are local to our GitHub repository. We recommend cloning
// the project from GitHub if you want to run this example.
// For more information, see https://github.com/awsdocs/aws-doc-sdk-examples.
import { dirnameFromMetaUrl } from "@aws-doc-sdk-examples/lib/utils/util-fs.js";
import { chunkArray } from "@aws-doc-sdk-examples/lib/utils/util-array.js";

const dirname = dirnameFromMetaUrl(import.meta.url);

const client = new DynamoDBClient({});
const docClient = DynamoDBDocumentClient.from(client);

export const main = async () => {
  const file = readFileSync(
    `${dirname}../../../../../resources/sample_files/movies.json`,
  );

  const movies = JSON.parse(file.toString());

  // chunkArray is a local convenience function. It takes an array and returns
  // a generator function. The generator function yields every N items.
  const movieChunks = chunkArray(movies, 25);

  // For every chunk of 25 movies, make one BatchWrite request.
  for (const chunk of movieChunks) {
    const putRequests = chunk.map((movie) => ({
      PutRequest: {
        Item: movie,
      },
    }));

    const command = new BatchWriteCommand({
      RequestItems: {
        // An existing table is required. A composite key of 'title' and 'year'
 is recommended
        // to account for duplicate titles.
        BatchWriteMoviesTable: putRequests,
```

```
      },
    });

    await docClient.send(command);
  }
};
```

- For API details, see BatchWriteItem in *AWS SDK for JavaScript API Reference*.

SDK for JavaScript (v2)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create DynamoDB service object
var ddb = new AWS.DynamoDB({ apiVersion: "2012-08-10" });

var params = {
  RequestItems: {
    TABLE_NAME: [
      {
        PutRequest: {
          Item: {
            KEY: { N: "KEY_VALUE" },
            ATTRIBUTE_1: { S: "ATTRIBUTE_1_VALUE" },
            ATTRIBUTE_2: { N: "ATTRIBUTE_2_VALUE" },
          },
        },
      },
      {
        PutRequest: {
          Item: {
            KEY: { N: "KEY_VALUE" },
            ATTRIBUTE_1: { S: "ATTRIBUTE_1_VALUE" },
```

```
            ATTRIBUTE_2: { N: "ATTRIBUTE_2_VALUE" },
          },
        },
      },
    ],
  },
};

ddb.batchWriteItem(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data);
  }
});
```

- For more information, see AWS SDK for JavaScript Developer Guide.
- For API details, see BatchWriteItem in *AWS SDK for JavaScript API Reference*.

PHP

SDK for PHP

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    public function writeBatch(string $TableName, array $Batch, int $depth = 2)
    {
        if (--$depth <= 0) {
            throw new Exception("Max depth exceeded. Please try with fewer batch
 items or increase depth.");
        }

        $marshal = new Marshaler();
        $total = 0;
        foreach (array_chunk($Batch, 25) as $Items) {
            foreach ($Items as $Item) {
```

```
                $BatchWrite['RequestItems'][$TableName][] = ['PutRequest' =>
 ['Item' => $marshal->marshalItem($Item)]];
            }
            try {
                echo "Batching another " . count($Items) . " for a total of " .
 ($total += count($Items)) . " items!\n";
                $response = $this->dynamoDbClient->batchWriteItem($BatchWrite);
                $BatchWrite = [];
            } catch (Exception $e) {
                echo "uh oh...";
                echo $e->getMessage();
                die();
            }
            if ($total >= 250) {
                echo "250 movies is probably enough. Right? We can stop there.
\n";
                break;
            }
        }
    }
```

- For API details, see BatchWriteItem in *AWS SDK for PHP API Reference*.

PowerShell

Tools for PowerShell V4

Example 1: Creates a new item, or replaces an existing item with a new item in the DynamoDB tables Music and Songs.

```
$item = @{
    SongTitle = 'Somewhere Down The Road'
    Artist = 'No One You Know'
    AlbumTitle = 'Somewhat Famous'
    Price = 1.94
    Genre = 'Country'
    CriticRating = 10.0
} | ConvertTo-DDBItem

$writeRequest = New-Object Amazon.DynamoDBv2.Model.WriteRequest
$writeRequest.PutRequest = [Amazon.DynamoDBv2.Model.PutRequest]$item
```

```
$requestItem = @{
    'Music' = [Amazon.DynamoDBv2.Model.WriteRequest]($writeRequest)
    'Songs' = [Amazon.DynamoDBv2.Model.WriteRequest]($writeRequest)
}

Set-DDBBatchItem -RequestItem $requestItem
```

- For API details, see BatchWriteItem in *AWS Tools for PowerShell Cmdlet Reference (V4)*.

Tools for PowerShell V5

Example 1: Creates a new item, or replaces an existing item with a new item in the DynamoDB tables Music and Songs.

```
$item = @{
    SongTitle = 'Somewhere Down The Road'
    Artist = 'No One You Know'
    AlbumTitle = 'Somewhat Famous'
    Price = 1.94
    Genre = 'Country'
    CriticRating = 10.0
} | ConvertTo-DDBItem

$writeRequest = New-Object Amazon.DynamoDBv2.Model.WriteRequest
$writeRequest.PutRequest = [Amazon.DynamoDBv2.Model.PutRequest]$item

$requestItem = @{
    'Music' = [Amazon.DynamoDBv2.Model.WriteRequest]($writeRequest)
    'Songs' = [Amazon.DynamoDBv2.Model.WriteRequest]($writeRequest)
}

Set-DDBBatchItem -RequestItem $requestItem
```

- For API details, see BatchWriteItem in *AWS Tools for PowerShell Cmdlet Reference (V5)*.

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
class Movies:
    """Encapsulates an Amazon DynamoDB table of movie data.

    Example data structure for a movie record in this table:
        {
            "year": 1999,
            "title": "For Love of the Game",
            "info": {
                "directors": ["Sam Raimi"],
                "release_date": "1999-09-15T00:00:00Z",
                "rating": 6.3,
                "plot": "A washed up pitcher flashes through his career.",
                "rank": 4987,
                "running_time_secs": 8220,
                "actors": [
                    "Kevin Costner",
                    "Kelly Preston",
                    "John C. Reilly"
                ]
            }
        }
    """

    def __init__(self, dyn_resource):
        """
        :param dyn_resource: A Boto3 DynamoDB resource.
        """
        self.dyn_resource = dyn_resource
        # The table variable is set during the scenario in the call to
        # 'exists' if the table exists. Otherwise, it is set by 'create_table'.
        self.table = None
```

```
    def write_batch(self, movies):
        """
        Fills an Amazon DynamoDB table with the specified data, using the Boto3
        Table.batch_writer() function to put the items in the table.
        Inside the context manager, Table.batch_writer builds a list of
        requests. On exiting the context manager, Table.batch_writer starts
sending
        batches of write requests to Amazon DynamoDB and automatically
        handles chunking, buffering, and retrying.

        :param movies: The data to put in the table. Each item must contain at
least
                       the keys required by the schema that was specified when
the
                       table was created.
        """
        try:
            with self.table.batch_writer() as writer:
                for movie in movies:
                    writer.put_item(Item=movie)
        except ClientError as err:
            logger.error(
                "Couldn't load data into table %s. Here's why: %s: %s",
                self.table.name,
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
```

- For API details, see BatchWriteItem in *AWS SDK for Python (Boto3) API Reference*.

Ruby

SDK for Ruby

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
class DynamoDBBasics
  attr_reader :dynamo_resource, :table

  def initialize(table_name)
    client = Aws::DynamoDB::Client.new(region: 'us-east-1')
    @dynamo_resource = Aws::DynamoDB::Resource.new(client: client)
    @table = @dynamo_resource.table(table_name)
  end

  # Fills an Amazon DynamoDB table with the specified data. Items are sent in
  # batches of 25 until all items are written.
  #
  # @param movies [Enumerable] The data to put in the table. Each item must
 contain at least
  #                            the keys required by the schema that was specified
 when the
  #                            table was created.
  def write_batch(movies)
    index = 0
    slice_size = 25
    while index < movies.length
      movie_items = []
      movies[index, slice_size].each do |movie|
        movie_items.append({ put_request: { item: movie } })
      end
      @dynamo_resource.client.batch_write_item({ request_items: { @table.name =>
 movie_items } })
      index += slice_size
    end
  rescue Aws::DynamoDB::Errors::ServiceError => e
    puts(
      "Couldn't load data into table #{@table.name}. Here's why:"
    )
    puts("\t#{e.code}: #{e.message}")
    raise
  end
```

- For API details, see [BatchWriteItem](#) in *AWS SDK for Ruby API Reference*.

Swift

SDK for Swift

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import AWSDynamoDB


    /// Populate the movie database from the specified JSON file.
    ///
    /// - Parameter jsonPath: Path to a JSON file containing movie data.
    ///
    func populate(jsonPath: String) async throws {
        do {
            guard let client = self.ddbClient else {
                throw MoviesError.UninitializedClient
            }

            // Create a Swift `URL` and use it to load the file into a `Data`
            // object. Then decode the JSON into an array of `Movie` objects.

            let fileUrl = URL(fileURLWithPath: jsonPath)
            let jsonData = try Data(contentsOf: fileUrl)

            var movieList = try JSONDecoder().decode([Movie].self, from:
 jsonData)

            // Truncate the list to the first 200 entries or so for this example.

            if movieList.count > 200 {
                movieList = Array(movieList[...199])
            }

            // Before sending records to the database, break the movie list into
            // 25-entry chunks, which is the maximum size of a batch item
 request.
```

```
            let count = movieList.count
            let chunks = stride(from: 0, to: count, by: 25).map {
                Array(movieList[$0 ..< Swift.min($0 + 25, count)])
            }

            // For each chunk, create a list of write request records and
populate
            // them with `PutRequest` requests, each specifying one movie from
the
            // chunk. Once the chunk's items are all in the `PutRequest` list,
            // send them to Amazon DynamoDB using the
            // `DynamoDBClient.batchWriteItem()` function.

            for chunk in chunks {
                var requestList: [DynamoDBClientTypes.WriteRequest] = []

                for movie in chunk {
                    let item = try await movie.getAsItem()
                    let request = DynamoDBClientTypes.WriteRequest(
                        putRequest: .init(
                            item: item
                        )
                    )
                    requestList.append(request)
                }

                let input = BatchWriteItemInput(requestItems: [tableName:
requestList])
                _ = try await client.batchWriteItem(input: input)
            }
        } catch {
            print("ERROR: populate:", dump(error))
            throw error
        }
    }
```

- For API details, see [BatchWriteItem](#) in *AWS SDK for Swift API reference*.

For a complete list of AWS SDK developer guides and code examples, see Using DynamoDB with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use `CreateTable` with an AWS SDK or CLI

The following code examples show how to use `CreateTable`.

Action examples are code excerpts from larger programs and must be run in context. You can see this action in context in the following code examples:

- Learn the basics
- Accelerate reads with DAX
- Advanced Global Secondary Index scenarios
- Create a table with global secondary index
- Create a table with warm throughput enabled
- Create and manage global tables demonstrating MREC
- Create and manage MRSC global tables
- Manage Global Secondary Indexes
- Manage resource-based policies
- Set up Attribute-Based Access Control
- Work with global tables and multi-Region replication eventual consistency (MREC)
- Work with Local Secondary Indexes
- Work with resource tagging
- Work with Streams and Time-to-Live
- Work with table encryption

.NET

SDK for .NET (v4)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    /// <summary>
    /// Creates a new Amazon DynamoDB table and then waits for the new
    /// table to become active.
    /// </summary>
    /// <param name="tableName">The name of the table to create.</param>
    /// <returns>A Boolean value indicating the success of the operation.</
returns>
    public async Task<bool> CreateMovieTableAsync(string tableName)
    {
        try
        {
            var response = await _amazonDynamoDB.CreateTableAsync(new
 CreateTableRequest
            {
                TableName = tableName,
                AttributeDefinitions = new List<AttributeDefinition>()
                {
                    new AttributeDefinition
                    {
                        AttributeName = "title",
                        AttributeType = ScalarAttributeType.S,
                    },
                    new AttributeDefinition
                    {
                        AttributeName = "year",
                        AttributeType = ScalarAttributeType.N,
                    },
                },
                KeySchema = new List<KeySchemaElement>()
                {
                    new KeySchemaElement
                    {
                        AttributeName = "year",
                        KeyType = KeyType.HASH,
                    },
                    new KeySchemaElement
                    {
                        AttributeName = "title",
                        KeyType = KeyType.RANGE,
                    },
                },
                BillingMode = BillingMode.PAY_PER_REQUEST,
```

```
            });

            // Wait until the table is ACTIVE and then report success.
            Console.Write("Waiting for table to become active...");

            var request = new DescribeTableRequest
            {
                TableName = response.TableDescription.TableName,
            };

            TableStatus status;

            int sleepDuration = 2000;

            do
            {
                Thread.Sleep(sleepDuration);

                var describeTableResponse = await
_amazonDynamoDB.DescribeTableAsync(request);
                status = describeTableResponse.Table.TableStatus;

                Console.Write(".");
            }
            while (status != "ACTIVE");

            return status == TableStatus.ACTIVE;
        }
        catch (ResourceInUseException ex)
        {
            Console.WriteLine($"Table {tableName} already exists. {ex.Message}");
            throw;
        }
        catch (AmazonDynamoDBException ex)
        {
            Console.WriteLine($"An Amazon DynamoDB error occurred while creating
table {tableName}. {ex.Message}");
            throw;
        }
        catch (Exception ex)
        {
            Console.WriteLine($"An error occurred while creating table
{tableName}. {ex.Message}");
            throw;
```

```
        }
    }
```

- For API details, see CreateTable in *AWS SDK for .NET API Reference*.

Bash

AWS CLI with Bash script

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
##############################################################################
# function dynamodb_create_table
#
# This function creates an Amazon DynamoDB table.
#
# Parameters:
#       -n table_name  -- The name of the table to create.
#       -a attribute_definitions -- JSON file path of a list of attributes and
 their types.
#       -k key_schema -- JSON file path of a list of attributes and their key
 types.
#
#  Returns:
#       0 - If successful.
#       1 - If it fails.
##############################################################################
function dynamodb_create_table() {
  local table_name attribute_definitions key_schema response
  local option OPTARG # Required to use getopts command in a function.

  ######################################
  # Function usage explanation
  ######################################
  function usage() {
    echo "function dynamodb_create_table"
```

```
   echo "Creates an Amazon DynamoDB table with on-demand billing."
   echo " -n table_name  -- The name of the table to create."
   echo " -a attribute_definitions -- JSON file path of a list of attributes and
their types."
   echo " -k key_schema -- JSON file path of a list of attributes and their key
types."
   echo ""
 }

 # Retrieve the calling parameters.
 while getopts "n:a:k:h" option; do
   case "${option}" in
     n) table_name="${OPTARG}" ;;
     a) attribute_definitions="${OPTARG}" ;;
     k) key_schema="${OPTARG}" ;;
     h)
       usage
       return 0
       ;;
     \?)
       echo "Invalid parameter"
       usage
       return 1
       ;;
   esac
 done
 export OPTIND=1

 if [[ -z "$table_name" ]]; then
   errecho "ERROR: You must provide a table name with the -n parameter."
   usage
   return 1
 fi

 if [[ -z "$attribute_definitions" ]]; then
   errecho "ERROR: You must provide an attribute definitions json file path the
-a parameter."
   usage
   return 1
 fi

 if [[ -z "$key_schema" ]]; then
   errecho "ERROR: You must provide a key schema json file path the -k
parameter."
```

```
    usage
    return 1
  fi

  iecho "Parameters:\n"
  iecho "    table_name:   $table_name"
  iecho "    attribute_definitions:   $attribute_definitions"
  iecho "    key_schema:   $key_schema"
  iecho ""

  response=$(aws dynamodb create-table \
    --table-name "$table_name" \
    --attribute-definitions file://"$attribute_definitions" \
    --billing-mode PAY_PER_REQUEST \
    --key-schema file://"$key_schema" )

  local error_code=${?}

  if [[ $error_code -ne 0 ]]; then
    aws_cli_error_log $error_code
    errecho "ERROR: AWS reports create-table operation failed.$response"
    return 1
  fi

  return 0
}
```

The utility functions used in this example.

```
###############################################################################
# function iecho
#
# This function enables the script to display the specified text only if
# the global variable $VERBOSE is set to true.
###############################################################################
function iecho() {
  if [[ $VERBOSE == true ]]; then
    echo "$@"
  fi
}

###############################################################################
```

```
# function errecho
#
# This function outputs everything sent to it to STDERR (standard error output).
#############################################################################
function errecho() {
  printf "%s\n" "$*" 1>&2
}

#############################################################################
# function aws_cli_error_log()
#
# This function is used to log the error messages from the AWS CLI.
#
# See https://docs.aws.amazon.com/cli/latest/topic/return-codes.html#cli-aws-
help-return-codes.
#
# The function expects the following argument:
#         $1 - The error code returned by the AWS CLI.
#
#  Returns:
#          0: - Success.
#
#############################################################################
function aws_cli_error_log() {
  local err_code=$1
  errecho "Error code : $err_code"
  if [ "$err_code" == 1 ]; then
    errecho "  One or more S3 transfers failed."
  elif [ "$err_code" == 2 ]; then
    errecho "  Command line failed to parse."
  elif [ "$err_code" == 130 ]; then
    errecho "  Process received SIGINT."
  elif [ "$err_code" == 252 ]; then
    errecho "  Command syntax invalid."
  elif [ "$err_code" == 253 ]; then
    errecho "  The system environment or configuration was invalid."
  elif [ "$err_code" == 254 ]; then
    errecho "  The service returned an error."
  elif [ "$err_code" == 255 ]; then
    errecho "  255 is a catch-all error."
  fi

  return 0
}
```

- For API details, see CreateTable in *AWS CLI Command Reference*.

C++

SDK for C++

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
//! Create an Amazon DynamoDB table.
/*!
  \sa createTable()
  \param tableName: Name for the DynamoDB table.
  \param primaryKey: Primary key for the DynamoDB table.
  \param clientConfiguration: AWS client configuration.
  \return bool: Function succeeded.
 */
bool AwsDoc::DynamoDB::createTable(const Aws::String &tableName,
                                   const Aws::String &primaryKey,
                                   const Aws::Client::ClientConfiguration
 &clientConfiguration) {
    Aws::DynamoDB::DynamoDBClient dynamoClient(clientConfiguration);

    std::cout << "Creating table " << tableName <<
              " with a simple primary key: \"" << primaryKey << "\"." <<
 std::endl;

    Aws::DynamoDB::Model::CreateTableRequest request;

    Aws::DynamoDB::Model::AttributeDefinition hashKey;
    hashKey.SetAttributeName(primaryKey);
    hashKey.SetAttributeType(Aws::DynamoDB::Model::ScalarAttributeType::S);
    request.AddAttributeDefinitions(hashKey);

    Aws::DynamoDB::Model::KeySchemaElement keySchemaElement;
    keySchemaElement.WithAttributeName(primaryKey).WithKeyType(
            Aws::DynamoDB::Model::KeyType::HASH);
```

```
    request.AddKeySchema(keySchemaElement);

    Aws::DynamoDB::Model::ProvisionedThroughput throughput;
    throughput.WithReadCapacityUnits(5).WithWriteCapacityUnits(5);
    request.SetProvisionedThroughput(throughput);
    request.SetTableName(tableName);

    const Aws::DynamoDB::Model::CreateTableOutcome &outcome =
 dynamoClient.CreateTable(
            request);
    if (outcome.IsSuccess()) {
        std::cout << "Table \""
                  << outcome.GetResult().GetTableDescription().GetTableName() <<
                  " created!" << std::endl;
    }
    else {
        std::cerr << "Failed to create table: " <<
 outcome.GetError().GetMessage()
                  << std::endl;
        return false;
    }

    return waitTableActive(tableName, dynamoClient);
}
```

Code that waits for the table to become active.

```
//! Query a newly created DynamoDB table until it is active.
/*!
  \sa waitTableActive()
  \param waitTableActive: The DynamoDB table's name.
  \param dynamoClient: A DynamoDB client.
  \return bool: Function succeeded.
*/
bool AwsDoc::DynamoDB::waitTableActive(const Aws::String &tableName,
                                       const Aws::DynamoDB::DynamoDBClient
 &dynamoClient) {

    // Repeatedly call DescribeTable until table is ACTIVE.
    const int MAX_QUERIES = 20;
    Aws::DynamoDB::Model::DescribeTableRequest request;
    request.SetTableName(tableName);
```

```
    int count = 0;
    while (count < MAX_QUERIES) {
        const Aws::DynamoDB::Model::DescribeTableOutcome &result =
 dynamoClient.DescribeTable(
                request);
        if (result.IsSuccess()) {
            Aws::DynamoDB::Model::TableStatus status =
 result.GetResult().GetTable().GetTableStatus();

            if (Aws::DynamoDB::Model::TableStatus::ACTIVE != status) {
                std::this_thread::sleep_for(std::chrono::seconds(1));
            }
            else {
                return true;
            }
        }
        else {
            std::cerr << "Error DynamoDB::waitTableActive "
                      << result.GetError().GetMessage() << std::endl;
            return false;
        }
        count++;
    }
    return false;
}
```

- For API details, see CreateTable in *AWS SDK for C++ API Reference*.

CLI

AWS CLI

Example 1: To create a table with tags

The following `create-table` example uses the specified attributes and key schema to create a table named `MusicCollection`. This table uses provisioned throughput and is encrypted at rest using the default AWS owned CMK. The command also applies a tag to the table, with a key of `Owner` and a value of `blueTeam`.

```
aws dynamodb create-table \
```

```
    --table-name MusicCollection \
    --attribute-
definitions AttributeName=Artist,AttributeType=S AttributeName=SongTitle,AttributeType=S
 \
    --key-
schema AttributeName=Artist,KeyType=HASH AttributeName=SongTitle,KeyType=RANGE \
    --provisioned-throughput ReadCapacityUnits=5,WriteCapacityUnits=5 \
    --tags Key=Owner,Value=blueTeam
```

Output:

```
{
    "TableDescription": {
        "AttributeDefinitions": [
            {
                "AttributeName": "Artist",
                "AttributeType": "S"
            },
            {
                "AttributeName": "SongTitle",
                "AttributeType": "S"
            }
        ],
        "ProvisionedThroughput": {
            "NumberOfDecreasesToday": 0,
            "WriteCapacityUnits": 5,
            "ReadCapacityUnits": 5
        },
        "TableSizeBytes": 0,
        "TableName": "MusicCollection",
        "TableStatus": "CREATING",
        "KeySchema": [
            {
                "KeyType": "HASH",
                "AttributeName": "Artist"
            },
            {
                "KeyType": "RANGE",
                "AttributeName": "SongTitle"
            }
        ],
        "ItemCount": 0,
        "CreationDateTime": "2020-05-26T16:04:41.627000-07:00",
```

```
        "TableArn": "arn:aws:dynamodb:us-west-2:123456789012:table/
MusicCollection",
        "TableId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111"
    }
}
```

For more information, see Basic Operations for Tables in the *Amazon DynamoDB Developer Guide*.

Example 2: To create a table in On-Demand Mode

The following example creates a table called `MusicCollection` using on-demand mode, rather than provisioned throughput mode. This is useful for tables with unpredictable workloads.

```
aws dynamodb create-table \
    --table-name MusicCollection \
    --attribute-
definitions AttributeName=Artist,AttributeType=S AttributeName=SongTitle,AttributeType=S
 \
    --key-
schema AttributeName=Artist,KeyType=HASH AttributeName=SongTitle,KeyType=RANGE \
    --billing-mode PAY_PER_REQUEST
```

Output:

```
{
    "TableDescription": {
        "AttributeDefinitions": [
            {
                "AttributeName": "Artist",
                "AttributeType": "S"
            },
            {
                "AttributeName": "SongTitle",
                "AttributeType": "S"
            }
        ],
        "TableName": "MusicCollection",
        "KeySchema": [
            {
                "AttributeName": "Artist",
```

```
                "KeyType": "HASH"
            },
            {
                "AttributeName": "SongTitle",
                "KeyType": "RANGE"
            }
        ],
        "TableStatus": "CREATING",
        "CreationDateTime": "2020-05-27T11:44:10.807000-07:00",
        "ProvisionedThroughput": {
            "NumberOfDecreasesToday": 0,
            "ReadCapacityUnits": 0,
            "WriteCapacityUnits": 0
        },
        "TableSizeBytes": 0,
        "ItemCount": 0,
        "TableArn": "arn:aws:dynamodb:us-west-2:123456789012:table/
MusicCollection",
        "TableId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111",
        "BillingModeSummary": {
            "BillingMode": "PAY_PER_REQUEST"
        }
    }
}
```

For more information, see Basic Operations for Tables in the *Amazon DynamoDB Developer Guide*.

Example 3: To create a table and encrypt it with a Customer Managed CMK

The following example creates a table named `MusicCollection` and encrypts it using a customer managed CMK.

```
aws dynamodb create-table \
    --table-name MusicCollection \
    --attribute-
definitions AttributeName=Artist,AttributeType=S AttributeName=SongTitle,AttributeType=S
 \
    --key-
schema AttributeName=Artist,KeyType=HASH AttributeName=SongTitle,KeyType=RANGE \
    --provisioned-throughput ReadCapacityUnits=5,WriteCapacityUnits=5 \
    --sse-specification Enabled=true,SSEType=KMS,KMSMasterKeyId=abcd1234-
abcd-1234-a123-ab1234a1b234
```

Output:

```
{
    "TableDescription": {
        "AttributeDefinitions": [
            {
                "AttributeName": "Artist",
                "AttributeType": "S"
            },
            {
                "AttributeName": "SongTitle",
                "AttributeType": "S"
            }
        ],
        "TableName": "MusicCollection",
        "KeySchema": [
            {
                "AttributeName": "Artist",
                "KeyType": "HASH"
            },
            {
                "AttributeName": "SongTitle",
                "KeyType": "RANGE"
            }
        ],
        "TableStatus": "CREATING",
        "CreationDateTime": "2020-05-27T11:12:16.431000-07:00",
        "ProvisionedThroughput": {
            "NumberOfDecreasesToday": 0,
            "ReadCapacityUnits": 5,
            "WriteCapacityUnits": 5
        },
        "TableSizeBytes": 0,
        "ItemCount": 0,
        "TableArn": "arn:aws:dynamodb:us-west-2:123456789012:table/
MusicCollection",
        "TableId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111",
        "SSEDescription": {
            "Status": "ENABLED",
            "SSEType": "KMS",
            "KMSMasterKeyArn": "arn:aws:kms:us-west-2:123456789012:key/abcd1234-
abcd-1234-a123-ab1234a1b234"
        }
    }
```

```
}
```

For more information, see Basic Operations for Tables in the *Amazon DynamoDB Developer Guide*.

Example 4: To create a table with a Local Secondary Index

The following example uses the specified attributes and key schema to create a table named `MusicCollection` with a Local Secondary Index named `AlbumTitleIndex`.

```
aws dynamodb create-table \
    --table-name MusicCollection \
    --attribute-
definitions AttributeName=Artist,AttributeType=S AttributeName=SongTitle,AttributeType=S
 \
    --key-
schema AttributeName=Artist,KeyType=HASH AttributeName=SongTitle,KeyType=RANGE \
    --provisioned-throughput ReadCapacityUnits=10,WriteCapacityUnits=5 \
    --local-secondary-indexes \
        "[
            {
                \"IndexName\": \"AlbumTitleIndex\",
                \"KeySchema\": [
                    {\"AttributeName\": \"Artist\",\"KeyType\":\"HASH\"},
                    {\"AttributeName\": \"AlbumTitle\",\"KeyType\":\"RANGE\"}
                ],
                \"Projection\": {
                    \"ProjectionType\": \"INCLUDE\",
                    \"NonKeyAttributes\": [\"Genre\", \"Year\"]
                }
            }
        ]"
```

Output:

```
{
    "TableDescription": {
        "AttributeDefinitions": [
            {
                "AttributeName": "AlbumTitle",
                "AttributeType": "S"
            },
            {
```

```
                "AttributeName": "Artist",
                "AttributeType": "S"
            },
            {
                "AttributeName": "SongTitle",
                "AttributeType": "S"
            }
        ],
        "TableName": "MusicCollection",
        "KeySchema": [
            {
                "AttributeName": "Artist",
                "KeyType": "HASH"
            },
            {
                "AttributeName": "SongTitle",
                "KeyType": "RANGE"
            }
        ],
        "TableStatus": "CREATING",
        "CreationDateTime": "2020-05-26T15:59:49.473000-07:00",
        "ProvisionedThroughput": {
            "NumberOfDecreasesToday": 0,
            "ReadCapacityUnits": 10,
            "WriteCapacityUnits": 5
        },
        "TableSizeBytes": 0,
        "ItemCount": 0,
        "TableArn": "arn:aws:dynamodb:us-west-2:123456789012:table/
MusicCollection",
        "TableId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111",
        "LocalSecondaryIndexes": [
            {
                "IndexName": "AlbumTitleIndex",
                "KeySchema": [
                    {
                        "AttributeName": "Artist",
                        "KeyType": "HASH"
                    },
                    {
                        "AttributeName": "AlbumTitle",
                        "KeyType": "RANGE"
                    }
                ],
```

```
                "Projection": {
                    "ProjectionType": "INCLUDE",
                    "NonKeyAttributes": [
                        "Genre",
                        "Year"
                    ]
                },
                "IndexSizeBytes": 0,
                "ItemCount": 0,
                "IndexArn": "arn:aws:dynamodb:us-west-2:123456789012:table/
MusicCollection/index/AlbumTitleIndex"
            }
        ]
    }
}
```

For more information, see Basic Operations for Tables in the *Amazon DynamoDB Developer Guide*.

Example 5: To create a table with a Global Secondary Index

The following example creates a table named `GameScores` with a Global Secondary Index called `GameTitleIndex`. The base table has a partition key of `UserId` and a sort key of `GameTitle`, allowing you to find an individual user's best score for a specific game efficiently, whereas the GSI has a partition key of `GameTitle` and a sort key of `TopScore`, allowing you to quickly find the overall highest score for a particular game.

```
aws dynamodb create-table \
    --table-name GameScores \
    --attribute-
definitions AttributeName=UserId,AttributeType=S AttributeName=GameTitle,AttributeType=S
 \
    --key-schema AttributeName=UserId,KeyType=HASH \
                 AttributeName=GameTitle,KeyType=RANGE \
    --provisioned-throughput ReadCapacityUnits=10,WriteCapacityUnits=5 \
    --global-secondary-indexes \
        "[
            {
                \"IndexName\": \"GameTitleIndex\",
                \"KeySchema\": [
                    {\"AttributeName\":\"GameTitle\",\"KeyType\":\"HASH\"},
                    {\"AttributeName\":\"TopScore\",\"KeyType\":\"RANGE\"}
                ],
```

```
                \"Projection\": {
                    \"ProjectionType\":\"INCLUDE\",
                    \"NonKeyAttributes\":[\"UserId\"]
                },
                \"ProvisionedThroughput\": {
                    \"ReadCapacityUnits\": 10,
                    \"WriteCapacityUnits\": 5
                }
            }
        ]"
```

Output:

```
{
    "TableDescription": {
        "AttributeDefinitions": [
            {
                "AttributeName": "GameTitle",
                "AttributeType": "S"
            },
            {
                "AttributeName": "TopScore",
                "AttributeType": "N"
            },
            {
                "AttributeName": "UserId",
                "AttributeType": "S"
            }
        ],
        "TableName": "GameScores",
        "KeySchema": [
            {
                "AttributeName": "UserId",
                "KeyType": "HASH"
            },
            {
                "AttributeName": "GameTitle",
                "KeyType": "RANGE"
            }
        ],
        "TableStatus": "CREATING",
        "CreationDateTime": "2020-05-26T17:28:15.602000-07:00",
        "ProvisionedThroughput": {
```

```
            "NumberOfDecreasesToday": 0,
            "ReadCapacityUnits": 10,
            "WriteCapacityUnits": 5
        },
        "TableSizeBytes": 0,
        "ItemCount": 0,
        "TableArn": "arn:aws:dynamodb:us-west-2:123456789012:table/GameScores",
        "TableId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111",
        "GlobalSecondaryIndexes": [
            {
                "IndexName": "GameTitleIndex",
                "KeySchema": [
                    {
                        "AttributeName": "GameTitle",
                        "KeyType": "HASH"
                    },
                    {
                        "AttributeName": "TopScore",
                        "KeyType": "RANGE"
                    }
                ],
                "Projection": {
                    "ProjectionType": "INCLUDE",
                    "NonKeyAttributes": [
                        "UserId"
                    ]
                },
                "IndexStatus": "CREATING",
                "ProvisionedThroughput": {
                    "NumberOfDecreasesToday": 0,
                    "ReadCapacityUnits": 10,
                    "WriteCapacityUnits": 5
                },
                "IndexSizeBytes": 0,
                "ItemCount": 0,
                "IndexArn": "arn:aws:dynamodb:us-west-2:123456789012:table/
GameScores/index/GameTitleIndex"
            }
        ]
    }
}
```

For more information, see Basic Operations for Tables in the *Amazon DynamoDB Developer Guide*.

Example 6: To create a table with multiple Global Secondary Indexes at once

The following example creates a table named `GameScores` with two Global Secondary Indexes. The GSI schemas are passed via a file, rather than on the command line.

```
aws dynamodb create-table \
    --table-name GameScores \
    --attribute-
definitions AttributeName=UserId,AttributeType=S AttributeName=GameTitle,AttributeType=S
 \
    --key-
schema AttributeName=UserId,KeyType=HASH AttributeName=GameTitle,KeyType=RANGE \
    --provisioned-throughput ReadCapacityUnits=10,WriteCapacityUnits=5 \
    --global-secondary-indexes file://gsi.json
```

Contents of `gsi.json`:

```
[
    {
        "IndexName": "GameTitleIndex",
        "KeySchema": [
            {
                "AttributeName": "GameTitle",
                "KeyType": "HASH"
            },
            {
                "AttributeName": "TopScore",
                "KeyType": "RANGE"
            }
        ],
        "Projection": {
            "ProjectionType": "ALL"
        },
        "ProvisionedThroughput": {
            "ReadCapacityUnits": 10,
            "WriteCapacityUnits": 5
        }
    },
    {
        "IndexName": "GameDateIndex",
```

```
        "KeySchema": [
            {
                "AttributeName": "GameTitle",
                "KeyType": "HASH"
            },
            {
                "AttributeName": "Date",
                "KeyType": "RANGE"
            }
        ],
        "Projection": {
            "ProjectionType": "ALL"
        },
        "ProvisionedThroughput": {
            "ReadCapacityUnits": 5,
            "WriteCapacityUnits": 5
        }
    }
]
```

Output:

```
{
    "TableDescription": {
        "AttributeDefinitions": [
            {
                "AttributeName": "Date",
                "AttributeType": "S"
            },
            {
                "AttributeName": "GameTitle",
                "AttributeType": "S"
            },
            {
                "AttributeName": "TopScore",
                "AttributeType": "N"
            },
            {
                "AttributeName": "UserId",
                "AttributeType": "S"
            }
        ],
        "TableName": "GameScores",
```

```
        "KeySchema": [
            {
                "AttributeName": "UserId",
                "KeyType": "HASH"
            },
            {
                "AttributeName": "GameTitle",
                "KeyType": "RANGE"
            }
        ],
        "TableStatus": "CREATING",
        "CreationDateTime": "2020-08-04T16:40:55.524000-07:00",
        "ProvisionedThroughput": {
            "NumberOfDecreasesToday": 0,
            "ReadCapacityUnits": 10,
            "WriteCapacityUnits": 5
        },
        "TableSizeBytes": 0,
        "ItemCount": 0,
        "TableArn": "arn:aws:dynamodb:us-west-2:123456789012:table/GameScores",
        "TableId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111",
        "GlobalSecondaryIndexes": [
            {
                "IndexName": "GameTitleIndex",
                "KeySchema": [
                    {
                        "AttributeName": "GameTitle",
                        "KeyType": "HASH"
                    },
                    {
                        "AttributeName": "TopScore",
                        "KeyType": "RANGE"
                    }
                ],
                "Projection": {
                    "ProjectionType": "ALL"
                },
                "IndexStatus": "CREATING",
                "ProvisionedThroughput": {
                    "NumberOfDecreasesToday": 0,
                    "ReadCapacityUnits": 10,
                    "WriteCapacityUnits": 5
                },
                "IndexSizeBytes": 0,
```

```
                "ItemCount": 0,
                "IndexArn": "arn:aws:dynamodb:us-west-2:123456789012:table/
GameScores/index/GameTitleIndex"
            },
            {
                "IndexName": "GameDateIndex",
                "KeySchema": [
                    {
                        "AttributeName": "GameTitle",
                        "KeyType": "HASH"
                    },
                    {
                        "AttributeName": "Date",
                        "KeyType": "RANGE"
                    }
                ],
                "Projection": {
                    "ProjectionType": "ALL"
                },
                "IndexStatus": "CREATING",
                "ProvisionedThroughput": {
                    "NumberOfDecreasesToday": 0,
                    "ReadCapacityUnits": 5,
                    "WriteCapacityUnits": 5
                },
                "IndexSizeBytes": 0,
                "ItemCount": 0,
                "IndexArn": "arn:aws:dynamodb:us-west-2:123456789012:table/
GameScores/index/GameDateIndex"
            }
        ]
    }
}
```

For more information, see Basic Operations for Tables in the *Amazon DynamoDB Developer Guide*.

Example 7: To create a table with Streams enabled

The following example creates a table called `GameScores` with DynamoDB Streams enabled. Both new and old images of each item will be written to the stream.

```
aws dynamodb create-table \
```

```
    --table-name GameScores \
    --attribute-
definitions AttributeName=UserId,AttributeType=S AttributeName=GameTitle,AttributeType=S
 \
    --key-
schema AttributeName=UserId,KeyType=HASH AttributeName=GameTitle,KeyType=RANGE \
    --provisioned-throughput ReadCapacityUnits=10,WriteCapacityUnits=5 \
    --stream-specification StreamEnabled=TRUE,StreamViewType=NEW_AND_OLD_IMAGES
```

Output:

```
{
    "TableDescription": {
        "AttributeDefinitions": [
            {
                "AttributeName": "GameTitle",
                "AttributeType": "S"
            },
            {
                "AttributeName": "UserId",
                "AttributeType": "S"
            }
        ],
        "TableName": "GameScores",
        "KeySchema": [
            {
                "AttributeName": "UserId",
                "KeyType": "HASH"
            },
            {
                "AttributeName": "GameTitle",
                "KeyType": "RANGE"
            }
        ],
        "TableStatus": "CREATING",
        "CreationDateTime": "2020-05-27T10:49:34.056000-07:00",
        "ProvisionedThroughput": {
            "NumberOfDecreasesToday": 0,
            "ReadCapacityUnits": 10,
            "WriteCapacityUnits": 5
        },
        "TableSizeBytes": 0,
        "ItemCount": 0,
```

```
        "TableArn": "arn:aws:dynamodb:us-west-2:123456789012:table/GameScores",
        "TableId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111",
        "StreamSpecification": {
            "StreamEnabled": true,
            "StreamViewType": "NEW_AND_OLD_IMAGES"
        },
        "LatestStreamLabel": "2020-05-27T17:49:34.056",
        "LatestStreamArn": "arn:aws:dynamodb:us-west-2:123456789012:table/
GameScores/stream/2020-05-27T17:49:34.056"
    }
}
```

For more information, see Basic Operations for Tables in the *Amazon DynamoDB Developer Guide*.

Example 8: To create a table with Keys-Only Stream enabled

The following example creates a table called `GameScores` with DynamoDB Streams enabled. Only the key attributes of modified items are written to the stream.

```
aws dynamodb create-table \
    --table-name GameScores \
    --attribute-
definitions AttributeName=UserId,AttributeType=S AttributeName=GameTitle,AttributeType=S
 \
    --key-
schema AttributeName=UserId,KeyType=HASH AttributeName=GameTitle,KeyType=RANGE \
    --provisioned-throughput ReadCapacityUnits=10,WriteCapacityUnits=5 \
    --stream-specification StreamEnabled=TRUE,StreamViewType=KEYS_ONLY
```

Output:

```
{
    "TableDescription": {
        "AttributeDefinitions": [
            {
                "AttributeName": "GameTitle",
                "AttributeType": "S"
            },
            {
                "AttributeName": "UserId",
                "AttributeType": "S"
```

```
            }
        ],
        "TableName": "GameScores",
        "KeySchema": [
            {
                "AttributeName": "UserId",
                "KeyType": "HASH"
            },
            {
                "AttributeName": "GameTitle",
                "KeyType": "RANGE"
            }
        ],
        "TableStatus": "CREATING",
        "CreationDateTime": "2023-05-25T18:45:34.140000+00:00",
        "ProvisionedThroughput": {
            "NumberOfDecreasesToday": 0,
            "ReadCapacityUnits": 10,
            "WriteCapacityUnits": 5
        },
        "TableSizeBytes": 0,
        "ItemCount": 0,
        "TableArn": "arn:aws:dynamodb:us-west-2:123456789012:table/GameScores",
        "TableId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111",
        "StreamSpecification": {
            "StreamEnabled": true,
            "StreamViewType": "KEYS_ONLY"
        },
        "LatestStreamLabel": "2023-05-25T18:45:34.140",
        "LatestStreamArn": "arn:aws:dynamodb:us-west-2:123456789012:table/
GameScores/stream/2023-05-25T18:45:34.140",
        "DeletionProtectionEnabled": false
    }
}
```

For more information, see Change data capture for DynamoDB Streams in the *Amazon DynamoDB Developer Guide*.

Example 9: To create a table with the Standard Infrequent Access class

The following example creates a table called `GameScores` and assigns the Standard-Infrequent Access (DynamoDB Standard-IA) table class. This table class is optimized for storage being the dominant cost.

```
aws dynamodb create-table \
    --table-name GameScores \
    --attribute-
definitions AttributeName=UserId,AttributeType=S AttributeName=GameTitle,AttributeType=S
 \
    --key-
schema AttributeName=UserId,KeyType=HASH AttributeName=GameTitle,KeyType=RANGE \
    --provisioned-throughput ReadCapacityUnits=10,WriteCapacityUnits=5 \
    --table-class STANDARD_INFREQUENT_ACCESS
```

Output:

```
{
    "TableDescription": {
        "AttributeDefinitions": [
            {
                "AttributeName": "GameTitle",
                "AttributeType": "S"
            },
            {
                "AttributeName": "UserId",
                "AttributeType": "S"
            }
        ],
        "TableName": "GameScores",
        "KeySchema": [
            {
                "AttributeName": "UserId",
                "KeyType": "HASH"
            },
            {
                "AttributeName": "GameTitle",
                "KeyType": "RANGE"
            }
        ],
        "TableStatus": "CREATING",
        "CreationDateTime": "2023-05-25T18:33:07.581000+00:00",
        "ProvisionedThroughput": {
            "NumberOfDecreasesToday": 0,
            "ReadCapacityUnits": 10,
            "WriteCapacityUnits": 5
        },
        "TableSizeBytes": 0,
```

```
        "ItemCount": 0,
        "TableArn": "arn:aws:dynamodb:us-west-2:123456789012:table/GameScores",
        "TableId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111",
        "TableClassSummary": {
            "TableClass": "STANDARD_INFREQUENT_ACCESS"
        },
        "DeletionProtectionEnabled": false
    }
}
```

For more information, see Table classes in the *Amazon DynamoDB Developer Guide*.

Example 10: To Create a table with Delete Protection enabled

The following example creates a table called `GameScores` and enables deletion protection.

```
aws dynamodb create-table \
    --table-name GameScores \
    --attribute-
definitions AttributeName=UserId,AttributeType=S AttributeName=GameTitle,AttributeType=S
 \
    --key-
schema AttributeName=UserId,KeyType=HASH AttributeName=GameTitle,KeyType=RANGE \
    --provisioned-throughput ReadCapacityUnits=10,WriteCapacityUnits=5 \
    --deletion-protection-enabled
```

Output:

```
{
    "TableDescription": {
        "AttributeDefinitions": [
            {
                "AttributeName": "GameTitle",
                "AttributeType": "S"
            },
            {
                "AttributeName": "UserId",
                "AttributeType": "S"
            }
        ],
        "TableName": "GameScores",
        "KeySchema": [
            {
```

```
                "AttributeName": "UserId",
                "KeyType": "HASH"
            },
            {
                "AttributeName": "GameTitle",
                "KeyType": "RANGE"
            }
        ],
        "TableStatus": "CREATING",
        "CreationDateTime": "2023-05-25T23:02:17.093000+00:00",
        "ProvisionedThroughput": {
            "NumberOfDecreasesToday": 0,
            "ReadCapacityUnits": 10,
            "WriteCapacityUnits": 5
        },
        "TableSizeBytes": 0,
        "ItemCount": 0,
        "TableArn": "arn:aws:dynamodb:us-west-2:123456789012:table/GameScores",
        "TableId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111",
        "DeletionProtectionEnabled": true
    }
}
```

For more information, see Using deletion protection in the *Amazon DynamoDB Developer Guide*.

- For API details, see CreateTable in *AWS CLI Command Reference*.

Go

SDK for Go V2

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import (
 "context"
 "errors"
```

```
 "log"
 "time"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/attributevalue"
 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/expression"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb/types"
)

// TableBasics encapsulates the Amazon DynamoDB service actions used in the
 examples.
// It contains a DynamoDB service client that is used to act on the specified
 table.
type TableBasics struct {
 DynamoDbClient *dynamodb.Client
 TableName      string
}



// CreateMovieTable creates a DynamoDB table with a composite primary key defined
 as
// a string sort key named `title`, and a numeric partition key named `year`.
// This function uses NewTableExistsWaiter to wait for the table to be created by
// DynamoDB before it returns.
func (basics TableBasics) CreateMovieTable(ctx context.Context)
 (*types.TableDescription, error) {
 var tableDesc *types.TableDescription
 table, err := basics.DynamoDbClient.CreateTable(ctx, &dynamodb.CreateTableInput{
  AttributeDefinitions: []types.AttributeDefinition{{
   AttributeName: aws.String("year"),
   AttributeType: types.ScalarAttributeTypeN,
  }, {
   AttributeName: aws.String("title"),
   AttributeType: types.ScalarAttributeTypeS,
  }},
  KeySchema: []types.KeySchemaElement{{
   AttributeName: aws.String("year"),
   KeyType:       types.KeyTypeHash,
  }, {
   AttributeName: aws.String("title"),
   KeyType:       types.KeyTypeRange,
  }},
```

```
  TableName:   aws.String(basics.TableName),
  BillingMode: types.BillingModePayPerRequest,
 })
 if err != nil {
  log.Printf("Couldn't create table %v. Here's why: %v\n", basics.TableName, err)
 } else {
  waiter := dynamodb.NewTableExistsWaiter(basics.DynamoDbClient)
  err = waiter.Wait(ctx, &dynamodb.DescribeTableInput{
   TableName: aws.String(basics.TableName)}, 5*time.Minute)
  if err != nil {
   log.Printf("Wait for table exists failed. Here's why: %v\n", err)
  }
  tableDesc = table.TableDescription
  log.Printf("Ccreating table test")
 }
 return tableDesc, err
}
```

- For API details, see CreateTable in *AWS SDK for Go API Reference*.

Java

SDK for Java 2.x

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import software.amazon.awssdk.core.waiters.WaiterResponse;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeDefinition;
import software.amazon.awssdk.services.dynamodb.model.BillingMode;
import software.amazon.awssdk.services.dynamodb.model.CreateTableRequest;
import software.amazon.awssdk.services.dynamodb.model.CreateTableResponse;
import software.amazon.awssdk.services.dynamodb.model.DescribeTableRequest;
import software.amazon.awssdk.services.dynamodb.model.DescribeTableResponse;
```

```
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.KeySchemaElement;
import software.amazon.awssdk.services.dynamodb.model.KeyType;
import software.amazon.awssdk.services.dynamodb.model.OnDemandThroughput;
import software.amazon.awssdk.services.dynamodb.model.ProvisionedThroughput;
import software.amazon.awssdk.services.dynamodb.model.ScalarAttributeType;
import software.amazon.awssdk.services.dynamodb.waiters.DynamoDbWaiter;

/**
 * Before running this Java V2 code example, set up your development
 * environment, including your credentials.
 * <p>
 * For more information, see the following documentation topic:
 * <p>
 * https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-
started.html
 */
public class CreateTable {
    public static void main(String[] args) {
        final String usage = """

            Usage:
                <tableName> <key>

            Where:
                tableName - The Amazon DynamoDB table to create (for example,
 Music3).
                key - The key for the Amazon DynamoDB table (for example,
 Artist).
            """;

        if (args.length != 2) {
            System.out.println(usage);
            System.exit(1);
        }

        String tableName = args[0];
        String key = args[1];
        System.out.println("Creating an Amazon DynamoDB table " + tableName + "
 with a simple primary key: " + key);
        Region region = Region.US_EAST_1;
        DynamoDbClient ddb = DynamoDbClient.builder()
            .region(region)
            .build();
```

```
        String result = createTable(ddb, tableName, key);
        System.out.println("New table is " + result);
        ddb.close();
    }

    public static String createTable(DynamoDbClient ddb, String tableName, String
 key) {
        DynamoDbWaiter dbWaiter = ddb.waiter();
        CreateTableRequest request = CreateTableRequest.builder()
            .attributeDefinitions(AttributeDefinition.builder()
                .attributeName(key)
                .attributeType(ScalarAttributeType.S)
                .build())
            .keySchema(KeySchemaElement.builder()
                .attributeName(key)
                .keyType(KeyType.HASH)
                .build())
            .billingMode(BillingMode.PAY_PER_REQUEST) //  DynamoDB automatically
 scales based on traffic.
            .tableName(tableName)
            .build();

        String newTable;
        try {
            CreateTableResponse response = ddb.createTable(request);
            DescribeTableRequest tableRequest = DescribeTableRequest.builder()
                .tableName(tableName)
                .build();

            // Wait until the Amazon DynamoDB table is created.
            WaiterResponse<DescribeTableResponse> waiterResponse =
 dbWaiter.waitUntilTableExists(tableRequest);
            waiterResponse.matched().response().ifPresent(System.out::println);
            newTable = response.tableDescription().tableName();
            return newTable;

        } catch (DynamoDbException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
        return "";
    }
}
```

- For API details, see CreateTable in *AWS SDK for Java 2.x API Reference*.

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import { CreateTableCommand, DynamoDBClient } from "@aws-sdk/client-dynamodb";

const client = new DynamoDBClient({});

export const main = async () => {
  const command = new CreateTableCommand({
    TableName: "EspressoDrinks",
    // For more information about data types,
    // see https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/
HowItWorks.NamingRulesDataTypes.html#HowItWorks.DataTypes and
    // https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/
Programming.LowLevelAPI.html#Programming.LowLevelAPI.DataTypeDescriptors
    AttributeDefinitions: [
      {
        AttributeName: "DrinkName",
        AttributeType: "S",
      },
    ],
    KeySchema: [
      {
        AttributeName: "DrinkName",
        KeyType: "HASH",
      },
    ],
    BillingMode: "PAY_PER_REQUEST",
  });

  const response = await client.send(command);
```

```
  console.log(response);
  return response;
};
```

- For more information, see AWS SDK for JavaScript Developer Guide.
- For API details, see CreateTable in *AWS SDK for JavaScript API Reference*.

SDK for JavaScript (v2)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create the DynamoDB service object
var ddb = new AWS.DynamoDB({ apiVersion: "2012-08-10" });

var params = {
  AttributeDefinitions: [
    {
      AttributeName: "CUSTOMER_ID",
      AttributeType: "N",
    },
    {
      AttributeName: "CUSTOMER_NAME",
      AttributeType: "S",
    },
  ],
  KeySchema: [
    {
      AttributeName: "CUSTOMER_ID",
      KeyType: "HASH",
    },
    {
      AttributeName: "CUSTOMER_NAME",
```

```
      KeyType: "RANGE",
    },
  ],
  ProvisionedThroughput: {
    ReadCapacityUnits: 1,
    WriteCapacityUnits: 1,
  },
  TableName: "CUSTOMER_LIST",
  StreamSpecification: {
    StreamEnabled: false,
  },
};

// Call DynamoDB to create the table
ddb.createTable(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Table Created", data);
  }
});
```

- For more information, see AWS SDK for JavaScript Developer Guide.
- For API details, see CreateTable in *AWS SDK for JavaScript API Reference*.

Kotlin

SDK for Kotlin

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
suspend fun createNewTable(
    tableNameVal: String,
    key: String,
): String? {
    val attDef =
```

```
        AttributeDefinition {
            attributeName = key
            attributeType = ScalarAttributeType.S
        }

    val keySchemaVal =
        KeySchemaElement {
            attributeName = key
            keyType = KeyType.Hash
        }

    val request =
        CreateTableRequest {
            attributeDefinitions = listOf(attDef)
            keySchema = listOf(keySchemaVal)
            billingMode = BillingMode.PayPerRequest
            tableName = tableNameVal
        }

    DynamoDbClient.fromEnvironment { region = "us-east-1" }.use { ddb ->
        var tableArn: String
        val response = ddb.createTable(request)
        ddb.waitUntilTableExists {
            // suspend call
            tableName = tableNameVal
        }
        tableArn = response.tableDescription!!.tableArn.toString()
        println("Table $tableArn is ready")
        return tableArn
    }
}
```

- For API details, see CreateTable in *AWS SDK for Kotlin API reference*.

PHP

SDK for PHP

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Create a table.

```
        $tableName = "ddb_demo_table_$uuid";
        $service->createTable(
            $tableName,
            [
                new DynamoDBAttribute('year', 'N', 'HASH'),
                new DynamoDBAttribute('title', 'S', 'RANGE')
            ]
        );

    public function createTable(string $tableName, array $attributes)
    {
        $keySchema = [];
        $attributeDefinitions = [];
        foreach ($attributes as $attribute) {
            if (is_a($attribute, DynamoDBAttribute::class)) {
                $keySchema[] = ['AttributeName' => $attribute->AttributeName,
'KeyType' => $attribute->KeyType];
                $attributeDefinitions[] =
                    ['AttributeName' => $attribute->AttributeName,
'AttributeType' => $attribute->AttributeType];
            }
        }

        $this->dynamoDbClient->createTable([
            'TableName' => $tableName,
            'KeySchema' => $keySchema,
            'AttributeDefinitions' => $attributeDefinitions,
            'ProvisionedThroughput' => ['ReadCapacityUnits' => 10,
'WriteCapacityUnits' => 10],
        ]);
    }
```

- For API details, see CreateTable in *AWS SDK for PHP API Reference*.

PowerShell

Tools for PowerShell V4

Example 1: This example creates a table named Thread that has a primary key consisting of 'ForumName' (key type hash) and 'Subject' (key type range). The schema used to construct the table can be piped into each cmdlet as shown or specified using the -Schema parameter.

```
$schema = New-DDBTableSchema
$schema | Add-DDBKeySchema -KeyName "ForumName" -KeyDataType "S"
$schema | Add-DDBKeySchema -KeyName "Subject" -KeyType RANGE -KeyDataType "S"
$schema | New-DDBTable -TableName "Thread" -ReadCapacity 10 -WriteCapacity 5
```

Output:

```
AttributeDefinitions   : {ForumName, Subject}
TableName              : Thread
KeySchema              : {ForumName, Subject}
TableStatus            : CREATING
CreationDateTime       : 10/28/2013 4:39:49 PM
ProvisionedThroughput  : Amazon.DynamoDBv2.Model.ProvisionedThroughputDescription
TableSizeBytes         : 0
ItemCount              : 0
LocalSecondaryIndexes  : {}
```

Example 2: This example creates a table named Thread that has a primary key consisting of 'ForumName' (key type hash) and 'Subject' (key type range). A local secondary index is also defined. The key of the local secondary index will be set automatically from the primary hash key on the table (ForumName). The schema used to construct the table can be piped into each cmdlet as shown or specified using the -Schema parameter.

```
$schema = New-DDBTableSchema
$schema | Add-DDBKeySchema -KeyName "ForumName" -KeyDataType "S"
$schema | Add-DDBKeySchema -KeyName "Subject" -KeyDataType "S"
$schema | Add-DDBIndexSchema -IndexName "LastPostIndex" -RangeKeyName
 "LastPostDateTime" -RangeKeyDataType "S" -ProjectionType "keys_only"
```

```
$schema | New-DDBTable -TableName "Thread" -ReadCapacity 10 -WriteCapacity 5
```

Output:

```
AttributeDefinitions   : {ForumName, LastPostDateTime, Subject}
TableName              : Thread
KeySchema              : {ForumName, Subject}
TableStatus            : CREATING
CreationDateTime       : 10/28/2013 4:39:49 PM
ProvisionedThroughput  : Amazon.DynamoDBv2.Model.ProvisionedThroughputDescription
TableSizeBytes         : 0
ItemCount              : 0
LocalSecondaryIndexes  : {LastPostIndex}
```

Example 3: This example shows how to use a single pipeline to create a table named Thread that has a primary key consisting of 'ForumName' (key type hash) and 'Subject' (key type range) and a local secondary index. The Add-DDBKeySchema and Add-DDBIndexSchema create a new TableSchema object for you if one is not supplied from the pipeline or the -Schema parameter.

```
New-DDBTableSchema |
  Add-DDBKeySchema -KeyName "ForumName" -KeyDataType "S" |
  Add-DDBKeySchema -KeyName "Subject" -KeyDataType "S" |
  Add-DDBIndexSchema -IndexName "LastPostIndex" `
                     -RangeKeyName "LastPostDateTime" `
                     -RangeKeyDataType "S" `
                     -ProjectionType "keys_only" |
  New-DDBTable -TableName "Thread" -ReadCapacity 10 -WriteCapacity 5
```

Output:

```
AttributeDefinitions   : {ForumName, LastPostDateTime, Subject}
TableName              : Thread
KeySchema              : {ForumName, Subject}
TableStatus            : CREATING
CreationDateTime       : 10/28/2013 4:39:49 PM
ProvisionedThroughput  : Amazon.DynamoDBv2.Model.ProvisionedThroughputDescription
TableSizeBytes         : 0
ItemCount              : 0
LocalSecondaryIndexes  : {LastPostIndex}
```

- For API details, see CreateTable in *AWS Tools for PowerShell Cmdlet Reference (V4)*.

Tools for PowerShell V5

Example 1: This example creates a table named Thread that has a primary key consisting of 'ForumName' (key type hash) and 'Subject' (key type range). The schema used to construct the table can be piped into each cmdlet as shown or specified using the -Schema parameter.

```
$schema = New-DDBTableSchema
$schema | Add-DDBKeySchema -KeyName "ForumName" -KeyDataType "S"
$schema | Add-DDBKeySchema -KeyName "Subject" -KeyType RANGE -KeyDataType "S"
$schema | New-DDBTable -TableName "Thread" -ReadCapacity 10 -WriteCapacity 5
```

Output:

```
AttributeDefinitions   : {ForumName, Subject}
TableName              : Thread
KeySchema              : {ForumName, Subject}
TableStatus            : CREATING
CreationDateTime       : 10/28/2013 4:39:49 PM
ProvisionedThroughput  : Amazon.DynamoDBv2.Model.ProvisionedThroughputDescription
TableSizeBytes         : 0
ItemCount              : 0
LocalSecondaryIndexes  : {}
```

Example 2: This example creates a table named Thread that has a primary key consisting of 'ForumName' (key type hash) and 'Subject' (key type range). A local secondary index is also defined. The key of the local secondary index will be set automatically from the primary hash key on the table (ForumName). The schema used to construct the table can be piped into each cmdlet as shown or specified using the -Schema parameter.

```
$schema = New-DDBTableSchema
$schema | Add-DDBKeySchema -KeyName "ForumName" -KeyDataType "S"
$schema | Add-DDBKeySchema -KeyName "Subject" -KeyDataType "S"
$schema | Add-DDBIndexSchema -IndexName "LastPostIndex" -RangeKeyName
 "LastPostDateTime" -RangeKeyDataType "S" -ProjectionType "keys_only"
$schema | New-DDBTable -TableName "Thread" -ReadCapacity 10 -WriteCapacity 5
```

Output:

```
AttributeDefinitions   : {ForumName, LastPostDateTime, Subject}
TableName              : Thread
KeySchema              : {ForumName, Subject}
TableStatus            : CREATING
CreationDateTime       : 10/28/2013 4:39:49 PM
ProvisionedThroughput  : Amazon.DynamoDBv2.Model.ProvisionedThroughputDescription
TableSizeBytes         : 0
ItemCount              : 0
LocalSecondaryIndexes  : {LastPostIndex}
```

Example 3: This example shows how to use a single pipeline to create a table named Thread that has a primary key consisting of 'ForumName' (key type hash) and 'Subject' (key type range) and a local secondary index. The Add-DDBKeySchema and Add-DDBIndexSchema create a new TableSchema object for you if one is not supplied from the pipeline or the -Schema parameter.

```
New-DDBTableSchema |
  Add-DDBKeySchema -KeyName "ForumName" -KeyDataType "S" |
  Add-DDBKeySchema -KeyName "Subject" -KeyDataType "S" |
  Add-DDBIndexSchema -IndexName "LastPostIndex" `
                     -RangeKeyName "LastPostDateTime" `
                     -RangeKeyDataType "S" `
                     -ProjectionType "keys_only" |
  New-DDBTable -TableName "Thread" -ReadCapacity 10 -WriteCapacity 5
```

Output:

```
AttributeDefinitions   : {ForumName, LastPostDateTime, Subject}
TableName              : Thread
KeySchema              : {ForumName, Subject}
TableStatus            : CREATING
CreationDateTime       : 10/28/2013 4:39:49 PM
ProvisionedThroughput  : Amazon.DynamoDBv2.Model.ProvisionedThroughputDescription
TableSizeBytes         : 0
ItemCount              : 0
LocalSecondaryIndexes  : {LastPostIndex}
```

- For API details, see CreateTable in *AWS Tools for PowerShell Cmdlet Reference (V5)*.

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Create a table for storing movie data.

```
class Movies:
    """Encapsulates an Amazon DynamoDB table of movie data.

    Example data structure for a movie record in this table:
        {
            "year": 1999,
            "title": "For Love of the Game",
            "info": {
                "directors": ["Sam Raimi"],
                "release_date": "1999-09-15T00:00:00Z",
                "rating": 6.3,
                "plot": "A washed up pitcher flashes through his career.",
                "rank": 4987,
                "running_time_secs": 8220,
                "actors": [
                    "Kevin Costner",
                    "Kelly Preston",
                    "John C. Reilly"
                ]
            }
        }
    """

    def __init__(self, dyn_resource):
        """
        :param dyn_resource: A Boto3 DynamoDB resource.
        """
        self.dyn_resource = dyn_resource
        # The table variable is set during the scenario in the call to
        # 'exists' if the table exists. Otherwise, it is set by 'create_table'.
        self.table = None
```

```
    def create_table(self, table_name):
        """
        Creates an Amazon DynamoDB table that can be used to store movie data.
        The table uses the release year of the movie as the partition key and the
        title as the sort key.

        :param table_name: The name of the table to create.
        :return: The newly created table.
        """
        try:
            self.table = self.dyn_resource.create_table(
                TableName=table_name,
                KeySchema=[
                    {"AttributeName": "year", "KeyType": "HASH"},  # Partition
key
                    {"AttributeName": "title", "KeyType": "RANGE"},  # Sort key
                ],
                AttributeDefinitions=[
                    {"AttributeName": "year", "AttributeType": "N"},
                    {"AttributeName": "title", "AttributeType": "S"},
                ],
                BillingMode='PAY_PER_REQUEST',
            )
            self.table.wait_until_exists()
        except ClientError as err:
            logger.error(
                "Couldn't create table %s. Here's why: %s: %s",
                table_name,
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
        else:
            return self.table
```

- For API details, see CreateTable in *AWS SDK for Python (Boto3) API Reference*.

Ruby

SDK for Ruby

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
# Encapsulates an Amazon DynamoDB table of movie data.
class Scaffold
  attr_reader :dynamo_resource, :table_name, :table

  def initialize(table_name)
    client = Aws::DynamoDB::Client.new(region: 'us-east-1')
    @dynamo_resource = Aws::DynamoDB::Resource.new(client: client)
    @table_name = table_name
    @table = nil
    @logger = Logger.new($stdout)
    @logger.level = Logger::DEBUG
  end

  # Creates an Amazon DynamoDB table that can be used to store movie data.
  # The table uses the release year of the movie as the partition key and the
  # title as the sort key.
  #
  # @param table_name [String] The name of the table to create.
  # @return [Aws::DynamoDB::Table] The newly created table.
  def create_table(table_name)
    @table = @dynamo_resource.create_table(
      table_name: table_name,
      key_schema: [
        { attribute_name: 'year', key_type: 'HASH' }, # Partition key
        { attribute_name: 'title', key_type: 'RANGE' } # Sort key
      ],
      attribute_definitions: [
        { attribute_name: 'year', attribute_type: 'N' },
        { attribute_name: 'title', attribute_type: 'S' }
      ],
      billing_mode: 'PAY_PER_REQUEST'
    )
```

```
    @dynamo_resource.client.wait_until(:table_exists, table_name: table_name)
    @table
  rescue Aws::DynamoDB::Errors::ServiceError => e
    @logger.error("Failed create table #{table_name}:\n#{e.code}: #{e.message}")
    raise
  end
```

- For API details, see CreateTable in *AWS SDK for Ruby API Reference*.

Rust

SDK for Rust

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
pub async fn create_table(
    client: &Client,
    table: &str,
    key: &str,
) -> Result<CreateTableOutput, Error> {
    let a_name: String = key.into();
    let table_name: String = table.into();

    let ad = AttributeDefinition::builder()
        .attribute_name(&a_name)
        .attribute_type(ScalarAttributeType::S)
        .build()
        .map_err(Error::BuildError)?;

    let ks = KeySchemaElement::builder()
        .attribute_name(&a_name)
        .key_type(KeyType::Hash)
        .build()
        .map_err(Error::BuildError)?;

    let create_table_response = client
        .create_table()
```

```
        .table_name(table_name)
        .key_schema(ks)
        .attribute_definitions(ad)
        .billing_mode(BillingMode::PayPerRequest)
        .send()
        .await;

    match create_table_response {
        Ok(out) => {
            println!("Added table {} with key {}", table, key);
            Ok(out)
        }
        Err(e) => {
            eprintln!("Got an error creating table:");
            eprintln!("{}", e);
            Err(Error::unhandled(e))
        }
    }
}
```

- For API details, see CreateTable in *AWS SDK for Rust API reference*.

SAP ABAP

SDK for SAP ABAP

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    TRY.
        DATA(lt_keyschema) = VALUE /aws1/cl_dynkeyschemaelement=>tt_keyschema(
          ( NEW /aws1/cl_dynkeyschemaelement( iv_attributename = 'year'
                                              iv_keytype = 'HASH' ) )
          ( NEW /aws1/cl_dynkeyschemaelement( iv_attributename = 'title'
                                              iv_keytype = 'RANGE' ) ) ).
        DATA(lt_attributedefinitions) = VALUE /aws1/
cl_dynattributedefn=>tt_attributedefinitions(
          ( NEW /aws1/cl_dynattributedefn( iv_attributename = 'year'
```

```
                                          iv_attributetype = 'N' ) )
          ( NEW /aws1/cl_dynattributedefn( iv_attributename = 'title'
                                          iv_attributetype = 'S' ) ) ).

        " Adjust read/write capacities as desired.
        DATA(lo_dynprovthroughput)  = NEW /aws1/cl_dynprovthroughput(
          iv_readcapacityunits = 5
          iv_writecapacityunits = 5 ).
        oo_result = lo_dyn->createtable(
          it_keyschema = lt_keyschema
          iv_tablename = iv_table_name
          it_attributedefinitions = lt_attributedefinitions
          io_provisionedthroughput = lo_dynprovthroughput ).
        " Table creation can take some time. Wait till table exists before
returning.
        lo_dyn->get_waiter( )->tableexists(
          iv_max_wait_time = 200
          iv_tablename     = iv_table_name ).
        MESSAGE 'DynamoDB Table' && iv_table_name && 'created.' TYPE 'I'.
        " This exception can happen if the table already exists.
      CATCH /aws1/cx_dynresourceinuseex INTO DATA(lo_resourceinuseex).
        DATA(lv_error) = |"{ lo_resourceinuseex->av_err_code }" -
{ lo_resourceinuseex->av_err_msg }|.
        MESSAGE lv_error TYPE 'E'.
    ENDTRY.
```

- For API details, see CreateTable in *AWS SDK for SAP ABAP API reference*.

Swift

SDK for Swift

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import AWSDynamoDB
```

```
    ///
    /// Create a movie table in the Amazon DynamoDB data store.
    ///
    private func createTable() async throws {
        do {
            guard let client = self.ddbClient else {
                throw MoviesError.UninitializedClient
            }

            let input = CreateTableInput(
                attributeDefinitions: [
                    DynamoDBClientTypes.AttributeDefinition(attributeName:
"year", attributeType: .n),
                    DynamoDBClientTypes.AttributeDefinition(attributeName:
"title", attributeType: .s)
                ],
                billingMode: DynamoDBClientTypes.BillingMode.payPerRequest,
                keySchema: [
                    DynamoDBClientTypes.KeySchemaElement(attributeName: "year",
keyType: .hash),
                    DynamoDBClientTypes.KeySchemaElement(attributeName: "title",
keyType: .range)
                ],
                tableName: self.tableName
            )
            let output = try await client.createTable(input: input)
            if output.tableDescription == nil {
                throw MoviesError.TableNotFound
            }
        } catch {
            print("ERROR: createTable:", dump(error))
            throw error
        }
    }
```

- For API details, see CreateTable in *AWS SDK for Swift API reference*.

For a complete list of AWS SDK developer guides and code examples, see Using DynamoDB with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use `DeleteItem` with an AWS SDK or CLI

The following code examples show how to use `DeleteItem`.

Action examples are code excerpts from larger programs and must be run in context. You can see this action in context in the following code examples:

- Learn the basics
- Use conditional operations

.NET

SDK for .NET (v4)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    /// <summary>
    /// Deletes a single item from a DynamoDB table.
    /// </summary>
    /// <param name="tableName">The name of the table from which the item
    /// will be deleted.</param>
    /// <param name="movieToDelete">A movie object containing the title and
    /// year of the movie to delete.</param>
    /// <returns>A Boolean value indicating the success or failure of the
    /// delete operation.</returns>
    public async Task<bool> DeleteItemAsync(
        string tableName,
        Movie movieToDelete)
    {
        try
        {
            var key = new Dictionary<string, AttributeValue>
            {
                ["title"] = new AttributeValue { S = movieToDelete.Title },
                ["year"] = new AttributeValue { N =
movieToDelete.Year.ToString() },
```

```
            };

            var request = new DeleteItemRequest { TableName = tableName, Key =
key, };

            await _amazonDynamoDB.DeleteItemAsync(request);
            return true;
        }
        catch (ResourceNotFoundException ex)
        {
            Console.WriteLine($"Table {tableName} was not found. {ex.Message}");
            return false;
        }
        catch (AmazonDynamoDBException ex)
        {
            Console.WriteLine($"An Amazon DynamoDB error occurred while deleting
item. {ex.Message}");
            throw;
        }
        catch (Exception ex)
        {
            Console.WriteLine($"An error occurred while deleting item.
{ex.Message}");
            throw;
        }
    }
```

- For API details, see DeleteItem in *AWS SDK for .NET API Reference*.

Bash

AWS CLI with Bash script

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
##############################################################################
```

```
# function dynamodb_delete_item
#
# This function deletes an item from a DynamoDB table.
#
# Parameters:
#       -n table_name  -- The name of the table.
#       -k keys  -- Path to json file containing the keys that identify the item
 to delete.
#
#  Returns:
#       0 - If successful.
#       1 - If it fails.
###############################################################################
function dynamodb_delete_item() {
  local table_name keys response
  local option OPTARG # Required to use getopts command in a function.

  # ######################################
  # Function usage explanation
  ######################################
  function usage() {
    echo "function dynamodb_delete_item"
    echo "Delete an item from a DynamoDB table."
    echo " -n table_name  -- The name of the table."
    echo " -k keys  -- Path to json file containing the keys that identify the
 item to delete."
    echo ""
  }
  while getopts "n:k:h" option; do
    case "${option}" in
      n) table_name="${OPTARG}" ;;
      k) keys="${OPTARG}" ;;
      h)
        usage
        return 0
        ;;
      \?)
        echo "Invalid parameter"
        usage
        return 1
        ;;
    esac
  done
  export OPTIND=1
```

```
  if [[ -z "$table_name" ]]; then
    errecho "ERROR: You must provide a table name with the -n parameter."
    usage
    return 1
  fi

  if [[ -z "$keys" ]]; then
    errecho "ERROR: You must provide a keys json file path the -k parameter."
    usage
    return 1
  fi

  iecho "Parameters:\n"
  iecho "    table_name:   $table_name"
  iecho "    keys:   $keys"
  iecho ""

  response=$(aws dynamodb delete-item \
    --table-name "$table_name" \
    --key file://"$keys")

  local error_code=${?}

  if [[ $error_code -ne 0 ]]; then
    aws_cli_error_log $error_code
    errecho "ERROR: AWS reports delete-item operation failed.$response"
    return 1
  fi

  return 0

}
```

The utility functions used in this example.

```
###############################################################################
# function iecho
#
# This function enables the script to display the specified text only if
# the global variable $VERBOSE is set to true.
###############################################################################
```

```
function iecho() {
  if [[ $VERBOSE == true ]]; then
    echo "$@"
  fi
}

#############################################################################
# function errecho
#
# This function outputs everything sent to it to STDERR (standard error output).
#############################################################################
function errecho() {
  printf "%s\n" "$*" 1>&2
}

#############################################################################
# function aws_cli_error_log()
#
# This function is used to log the error messages from the AWS CLI.
#
# See https://docs.aws.amazon.com/cli/latest/topic/return-codes.html#cli-aws-
help-return-codes.
#
# The function expects the following argument:
#         $1 - The error code returned by the AWS CLI.
#
#  Returns:
#          0: - Success.
#
#############################################################################
function aws_cli_error_log() {
  local err_code=$1
  errecho "Error code : $err_code"
  if [ "$err_code" == 1 ]; then
    errecho "  One or more S3 transfers failed."
  elif [ "$err_code" == 2 ]; then
    errecho "  Command line failed to parse."
  elif [ "$err_code" == 130 ]; then
    errecho "  Process received SIGINT."
  elif [ "$err_code" == 252 ]; then
    errecho "  Command syntax invalid."
  elif [ "$err_code" == 253 ]; then
    errecho "  The system environment or configuration was invalid."
  elif [ "$err_code" == 254 ]; then
```

```
    errecho "  The service returned an error."
  elif [ "$err_code" == 255 ]; then
    errecho "  255 is a catch-all error."
  fi

  return 0
}
```

- For API details, see DeleteItem in *AWS CLI Command Reference*.

C++

SDK for C++

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
//! Delete an item from an Amazon DynamoDB table.
/*!
  \sa deleteItem()
  \param tableName: The table name.
  \param partitionKey: The partition key.
  \param partitionValue: The value for the partition key.
  \param clientConfiguration: AWS client configuration.
  \return bool: Function succeeded.
 */

bool AwsDoc::DynamoDB::deleteItem(const Aws::String &tableName,
                                  const Aws::String &partitionKey,
                                  const Aws::String &partitionValue,
                                  const Aws::Client::ClientConfiguration
 &clientConfiguration) {
    Aws::DynamoDB::DynamoDBClient dynamoClient(clientConfiguration);

    Aws::DynamoDB::Model::DeleteItemRequest request;

    request.AddKey(partitionKey,
                   Aws::DynamoDB::Model::AttributeValue().SetS(partitionValue));
```

```
    request.SetTableName(tableName);

    const Aws::DynamoDB::Model::DeleteItemOutcome &outcome =
 dynamoClient.DeleteItem(
            request);
    if (outcome.IsSuccess()) {
        std::cout << "Item \"" << partitionValue << "\" deleted!" << std::endl;
    }
    else {
        std::cerr << "Failed to delete item: " << outcome.GetError().GetMessage()
                  << std::endl;
        return false;
    }

    return waitTableActive(tableName, dynamoClient);
}
```

Code that waits for the table to become active.

```
//! Query a newly created DynamoDB table until it is active.
/*!
  \sa waitTableActive()
  \param waitTableActive: The DynamoDB table's name.
  \param dynamoClient: A DynamoDB client.
  \return bool: Function succeeded.
*/
bool AwsDoc::DynamoDB::waitTableActive(const Aws::String &tableName,
                                       const Aws::DynamoDB::DynamoDBClient
 &dynamoClient) {

    // Repeatedly call DescribeTable until table is ACTIVE.
    const int MAX_QUERIES = 20;
    Aws::DynamoDB::Model::DescribeTableRequest request;
    request.SetTableName(tableName);

    int count = 0;
    while (count < MAX_QUERIES) {
        const Aws::DynamoDB::Model::DescribeTableOutcome &result =
 dynamoClient.DescribeTable(
                request);
        if (result.IsSuccess()) {
```

```
            Aws::DynamoDB::Model::TableStatus status =
 result.GetResult().GetTable().GetTableStatus();

            if (Aws::DynamoDB::Model::TableStatus::ACTIVE != status) {
                std::this_thread::sleep_for(std::chrono::seconds(1));
            }
            else {
                return true;
            }
        }
        else {
            std::cerr << "Error DynamoDB::waitTableActive "
                      << result.GetError().GetMessage() << std::endl;
            return false;
        }
        count++;
    }
    return false;
}
```

- For API details, see DeleteItem in *AWS SDK for C++ API Reference*.

CLI

AWS CLI

Example 1: To delete an item

The following `delete-item` example deletes an item from the `MusicCollection` table and requests details about the item that was deleted and the capacity used by the request.

```
aws dynamodb delete-item \
    --table-name MusicCollection \
    --key file://key.json \
    --return-values ALL_OLD \
    --return-consumed-capacity TOTAL \
    --return-item-collection-metrics SIZE
```

Contents of `key.json`:

```
{
```

```
    "Artist": {"S": "No One You Know"},
    "SongTitle": {"S": "Scared of My Shadow"}
}
```

Output:

```
{
    "Attributes": {
        "AlbumTitle": {
            "S": "Blue Sky Blues"
        },
        "Artist": {
            "S": "No One You Know"
        },
        "SongTitle": {
            "S": "Scared of My Shadow"
        }
    },
    "ConsumedCapacity": {
        "TableName": "MusicCollection",
        "CapacityUnits": 2.0
    },
    "ItemCollectionMetrics": {
        "ItemCollectionKey": {
            "Artist": {
                "S": "No One You Know"
            }
        },
        "SizeEstimateRangeGB": [
            0.0,
            1.0
        ]
    }
}
```

For more information, see Writing an Item in the *Amazon DynamoDB Developer Guide*.

Example 2: To delete an item conditionally

The following example deletes an item from the `ProductCatalog` table only if its `ProductCategory` is either `Sporting Goods` or `Gardening Supplies` and its price is between 500 and 600. It returns details about the item that was deleted.

```
aws dynamodb delete-item \
    --table-name ProductCatalog \
    --key '{"Id":{"N":"456"}}' \
    --condition-expression "(ProductCategory IN (:cat1, :cat2)) and (#P
 between :lo and :hi)" \
    --expression-attribute-names file://names.json \
    --expression-attribute-values file://values.json \
    --return-values ALL_OLD
```

Contents of `names.json`:

```
{
    "#P": "Price"
}
```

Contents of `values.json`:

```
{
    ":cat1": {"S": "Sporting Goods"},
    ":cat2": {"S": "Gardening Supplies"},
    ":lo": {"N": "500"},
    ":hi": {"N": "600"}
}
```

Output:

```
{
    "Attributes": {
        "Id": {
            "N": "456"
        },
        "Price": {
            "N": "550"
        },
        "ProductCategory": {
            "S": "Sporting Goods"
        }
    }
}
```

For more information, see Writing an Item in the *Amazon DynamoDB Developer Guide*.

- For API details, see DeleteItem in *AWS CLI Command Reference*.

Go

SDK for Go V2

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import (
 "context"
 "errors"
 "log"
 "time"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/attributevalue"
 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/expression"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb/types"
)

// TableBasics encapsulates the Amazon DynamoDB service actions used in the
 examples.
// It contains a DynamoDB service client that is used to act on the specified
 table.
type TableBasics struct {
 DynamoDbClient *dynamodb.Client
 TableName      string
}



// DeleteMovie removes a movie from the DynamoDB table.
func (basics TableBasics) DeleteMovie(ctx context.Context, movie Movie) error {
 _, err := basics.DynamoDbClient.DeleteItem(ctx, &dynamodb.DeleteItemInput{
  TableName: aws.String(basics.TableName), Key: movie.GetKey(),
 })
```

```
 if err != nil {
  log.Printf("Couldn't delete %v from the table. Here's why: %v\n", movie.Title,
 err)
 }
 return err
}
```

Define a Movie struct that is used in this example.

```
import (
 "archive/zip"
 "bytes"
 "encoding/json"
 "fmt"
 "io"
 "log"
 "net/http"

 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/attributevalue"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb/types"
)

// Movie encapsulates data about a movie. Title and Year are the composite
 primary key
// of the movie in Amazon DynamoDB. Title is the sort key, Year is the partition
 key,
// and Info is additional data.
type Movie struct {
 Title string                 `dynamodbav:"title"`
 Year  int                    `dynamodbav:"year"`
 Info  map[string]interface{} `dynamodbav:"info"`
}

// GetKey returns the composite primary key of the movie in a format that can be
// sent to DynamoDB.
func (movie Movie) GetKey() map[string]types.AttributeValue {
 title, err := attributevalue.Marshal(movie.Title)
 if err != nil {
  panic(err)
 }
```

```
 year, err := attributevalue.Marshal(movie.Year)
 if err != nil {
  panic(err)
 }
 return map[string]types.AttributeValue{"title": title, "year": year}
}

// String returns the title, year, rating, and plot of a movie, formatted for the
 example.
func (movie Movie) String() string {
 return fmt.Sprintf("%v\n\tReleased: %v\n\tRating: %v\n\tPlot: %v\n",
  movie.Title, movie.Year, movie.Info["rating"], movie.Info["plot"])
}
```

- For API details, see DeleteItem in *AWS SDK for Go API Reference*.

Java

SDK for Java 2.x

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.DeleteItemRequest;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import java.util.HashMap;

/**
 * Before running this Java V2 code example, set up your development
 * environment, including your credentials.
 *
 * For more information, see the following documentation topic:
 *
```

```
 * https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-
started.html
 */
public class DeleteItem {
    public static void main(String[] args) {
        final String usage = """

                Usage:
                    <tableName> <key> <keyval>

                Where:
                    tableName - The Amazon DynamoDB table to delete the item from
 (for example, Music3).
                    key - The key used in the Amazon DynamoDB table (for example,
 Artist).\s
                    keyval - The key value that represents the item to delete
 (for example, Famous Band).
                """;

        if (args.length != 3) {
            System.out.println(usage);
            System.exit(1);
        }

        String tableName = args[0];
        String key = args[1];
        String keyVal = args[2];
        System.out.format("Deleting item \"%s\" from %s\n", keyVal, tableName);
        Region region = Region.US_EAST_1;
        DynamoDbClient ddb = DynamoDbClient.builder()
                .region(region)
                .build();

        deleteDynamoDBItem(ddb, tableName, key, keyVal);
        ddb.close();
    }

    public static void deleteDynamoDBItem(DynamoDbClient ddb, String tableName,
 String key, String keyVal) {
        HashMap<String, AttributeValue> keyToGet = new HashMap<>();
        keyToGet.put(key, AttributeValue.builder()
                .s(keyVal)
                .build());
```

```
        DeleteItemRequest deleteReq = DeleteItemRequest.builder()
                .tableName(tableName)
                .key(keyToGet)
                .build();

        try {
            ddb.deleteItem(deleteReq);
        } catch (DynamoDbException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }
}
```

- For API details, see DeleteItem in *AWS SDK for Java 2.x API Reference*.

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

This example uses the document client to simplify working with items in DynamoDB. For API details see DeleteCommand.

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
import { DynamoDBDocumentClient, DeleteCommand } from "@aws-sdk/lib-dynamodb";

const client = new DynamoDBClient({});
const docClient = DynamoDBDocumentClient.from(client);

export const main = async () => {
  const command = new DeleteCommand({
    TableName: "Sodas",
    Key: {
      Flavor: "Cola",
    },
```

```
  });

  const response = await docClient.send(command);
  console.log(response);
  return response;
};
```

- For more information, see AWS SDK for JavaScript Developer Guide.
- For API details, see DeleteItem in *AWS SDK for JavaScript API Reference*.

SDK for JavaScript (v2)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Delete an item from a table.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create the DynamoDB service object
var ddb = new AWS.DynamoDB({ apiVersion: "2012-08-10" });

var params = {
  TableName: "TABLE",
  Key: {
    KEY_NAME: { N: "VALUE" },
  },
};

// Call DynamoDB to delete the item from the table
ddb.deleteItem(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data);
  }
```

```
});
```

Delete an item from a table using the DynamoDB document client.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create DynamoDB document client
var docClient = new AWS.DynamoDB.DocumentClient({ apiVersion: "2012-08-10" });

var params = {
  Key: {
    HASH_KEY: VALUE,
  },
  TableName: "TABLE",
};

docClient.delete(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data);
  }
});
```

- For more information, see AWS SDK for JavaScript Developer Guide.
- For API details, see DeleteItem in *AWS SDK for JavaScript API Reference*.

Kotlin

SDK for Kotlin

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
suspend fun deleteDynamoDBItem(
    tableNameVal: String,
    keyName: String,
    keyVal: String,
) {
    val keyToGet = mutableMapOf<String, AttributeValue>()
    keyToGet[keyName] = AttributeValue.S(keyVal)

    val request =
        DeleteItemRequest {
            tableName = tableNameVal
            key = keyToGet
        }

    DynamoDbClient.fromEnvironment { region = "us-east-1" }.use { ddb ->
        ddb.deleteItem(request)
        println("Item with key matching $keyVal was deleted")
    }
}
```

- For API details, see DeleteItem in *AWS SDK for Kotlin API reference*.

PHP

SDK for PHP

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
        $key = [
            'Item' => [
                'title' => [
                    'S' => $movieName,
                ],
                'year' => [
                    'N' => $movieYear,
                ],
```

```
            ]
        ];

        $service->deleteItemByKey($tableName, $key);
        echo "But, bad news, this was a trap. That movie has now been deleted
 because of your rating...harsh.\n";

    public function deleteItemByKey(string $tableName, array $key)
    {
        $this->dynamoDbClient->deleteItem([
            'Key' => $key['Item'],
            'TableName' => $tableName,
        ]);
    }
```

- For API details, see DeleteItem in *AWS SDK for PHP API Reference*.

PowerShell

Tools for PowerShell V4

Example 1: Removes the DynamoDB item that matches the provided key.

```
$key = @{
    SongTitle = 'Somewhere Down The Road'
    Artist = 'No One You Know'
} | ConvertTo-DDBItem
Remove-DDBItem -TableName 'Music' -Key $key -Confirm:$false
```

- For API details, see DeleteItem in *AWS Tools for PowerShell Cmdlet Reference (V4)*.

Tools for PowerShell V5

Example 1: Removes the DynamoDB item that matches the provided key.

```
$key = @{
    SongTitle = 'Somewhere Down The Road'
    Artist = 'No One You Know'
} | ConvertTo-DDBItem
Remove-DDBItem -TableName 'Music' -Key $key -Confirm:$false
```

- For API details, see DeleteItem in *AWS Tools for PowerShell Cmdlet Reference (V5)*.

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
class Movies:
    """Encapsulates an Amazon DynamoDB table of movie data.

    Example data structure for a movie record in this table:
        {
            "year": 1999,
            "title": "For Love of the Game",
            "info": {
                "directors": ["Sam Raimi"],
                "release_date": "1999-09-15T00:00:00Z",
                "rating": 6.3,
                "plot": "A washed up pitcher flashes through his career.",
                "rank": 4987,
                "running_time_secs": 8220,
                "actors": [
                    "Kevin Costner",
                    "Kelly Preston",
                    "John C. Reilly"
                ]
            }
        }
    """

    def __init__(self, dyn_resource):
        """
        :param dyn_resource: A Boto3 DynamoDB resource.
        """
        self.dyn_resource = dyn_resource
        # The table variable is set during the scenario in the call to
        # 'exists' if the table exists. Otherwise, it is set by 'create_table'.
        self.table = None
```

```
    def delete_movie(self, title, year):
        """
        Deletes a movie from the table.

        :param title: The title of the movie to delete.
        :param year: The release year of the movie to delete.
        """
        try:
            self.table.delete_item(Key={"year": year, "title": title})
        except ClientError as err:
            logger.error(
                "Couldn't delete movie %s. Here's why: %s: %s",
                title,
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
```

You can specify a condition so that an item is deleted only when it meets certain criteria.

```
class UpdateQueryWrapper:
    def __init__(self, table):
        self.table = table


    def delete_underrated_movie(self, title, year, rating):
        """
        Deletes a movie only if it is rated below a specified value. By using a
        condition expression in a delete operation, you can specify that an item
 is
        deleted only when it meets certain criteria.

        :param title: The title of the movie to delete.
        :param year: The release year of the movie to delete.
        :param rating: The rating threshold to check before deleting the movie.
        """
        try:
            self.table.delete_item(
                Key={"year": year, "title": title},
                ConditionExpression="info.rating <= :val",
```

```
                ExpressionAttributeValues={":val": Decimal(str(rating))},
            )
        except ClientError as err:
            if err.response["Error"]["Code"] ==
 "ConditionalCheckFailedException":
                logger.warning(
                    "Didn't delete %s because its rating is greater than %s.",
                    title,
                    rating,
                )
            else:
                logger.error(
                    "Couldn't delete movie %s. Here's why: %s: %s",
                    title,
                    err.response["Error"]["Code"],
                    err.response["Error"]["Message"],
                )
            raise
```

- For API details, see DeleteItem in *AWS SDK for Python (Boto3) API Reference*.

Ruby

SDK for Ruby

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
class DynamoDBBasics
  attr_reader :dynamo_resource, :table

  def initialize(table_name)
    client = Aws::DynamoDB::Client.new(region: 'us-east-1')
    @dynamo_resource = Aws::DynamoDB::Resource.new(client: client)
    @table = @dynamo_resource.table(table_name)
  end
```

```
  # Deletes a movie from the table.
  #
  # @param title [String] The title of the movie to delete.
  # @param year [Integer] The release year of the movie to delete.
  def delete_item(title, year)
    @table.delete_item(key: { 'year' => year, 'title' => title })
  rescue Aws::DynamoDB::Errors::ServiceError => e
    puts("Couldn't delete movie #{title}. Here's why:")
    puts("\t#{e.code}: #{e.message}")
    raise
  end
```

- For API details, see DeleteItem in *AWS SDK for Ruby API Reference*.

Rust

SDK for Rust

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
pub async fn delete_item(
    client: &Client,
    table: &str,
    key: &str,
    value: &str,
) -> Result<DeleteItemOutput, Error> {
    match client
        .delete_item()
        .table_name(table)
        .key(key, AttributeValue::S(value.into()))
        .send()
        .await
    {
        Ok(out) => {
            println!("Deleted item from table");
            Ok(out)
        }
```

```
        Err(e) => Err(Error::unhandled(e)),
    }
}
```

- For API details, see DeleteItem in *AWS SDK for Rust API reference*.

SAP ABAP

SDK for SAP ABAP

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    TRY.
        DATA(lo_resp) = lo_dyn->deleteitem(
          iv_tablename                = iv_table_name
          it_key                      = it_key_input ).
        MESSAGE 'Deleted one item.' TYPE 'I'.
      CATCH /aws1/cx_dyncondalcheckfaile00.
        MESSAGE 'A condition specified in the operation could not be evaluated.'
TYPE 'E'.
      CATCH /aws1/cx_dynresourcenotfoundex.
        MESSAGE 'The table or index does not exist' TYPE 'E'.
      CATCH /aws1/cx_dyntransactconflictex.
        MESSAGE 'Another transaction is using the item' TYPE 'E'.
    ENDTRY.
```

- For API details, see DeleteItem in *AWS SDK for SAP ABAP API reference*.

Swift

SDK for Swift

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import AWSDynamoDB


    /// Delete a movie, given its title and release year.
    ///
    /// - Parameters:
    ///   - title: The movie's title.
    ///   - year: The movie's release year.
    ///
    func delete(title: String, year: Int) async throws {
        do {
            guard let client = self.ddbClient else {
                throw MoviesError.UninitializedClient
            }

            let input = DeleteItemInput(
                key: [
                    "year": .n(String(year)),
                    "title": .s(title)
                ],
                tableName: self.tableName
            )
            _ = try await client.deleteItem(input: input)
        } catch {
            print("ERROR: delete:", dump(error))
            throw error
        }
    }
```

- For API details, see DeleteItem in *AWS SDK for Swift API reference*.

For a complete list of AWS SDK developer guides and code examples, see Using DynamoDB with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use `DeleteTable` with an AWS SDK or CLI

The following code examples show how to use `DeleteTable`.

Action examples are code excerpts from larger programs and must be run in context. You can see this action in context in the following code examples:

- Learn the basics
- Accelerate reads with DAX
- Create and manage MRSC global tables
- Manage Global Secondary Indexes
- Work with global tables and multi-Region replication eventual consistency (MREC)

.NET

SDK for .NET (v4)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    /// <summary>
    /// Deletes a DynamoDB table.
    /// </summary>
    /// <param name="tableName">The name of the table to delete.</param>
    /// <returns>A Boolean value indicating the success of the operation.</
returns>
    public async Task<bool> DeleteTableAsync(string tableName)
    {
        try
        {
            var request = new DeleteTableRequest
            {
                TableName = tableName,
```

```
            };

            var response = await _amazonDynamoDB.DeleteTableAsync(request);

            Console.WriteLine($"Table {response.TableDescription.TableName}
successfully deleted.");
            return true;

        }
        catch (ResourceNotFoundException ex)
        {
            Console.WriteLine($"Table {tableName} was not found and cannot be
deleted. {ex.Message}");
            return false;
        }
        catch (AmazonDynamoDBException ex)
        {
            Console.WriteLine($"An Amazon DynamoDB error occurred while deleting
table {tableName}. {ex.Message}");
            return false;
        }
        catch (Exception ex)
        {
            Console.WriteLine($"An error occurred while deleting table
{tableName}. {ex.Message}");
            return false;
        }
    }
```

- For API details, see DeleteTable in *AWS SDK for .NET API Reference*.

Bash

AWS CLI with Bash script

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
###############################################################################
# function dynamodb_delete_table
#
# This function deletes a DynamoDB table.
#
# Parameters:
#       -n table_name  -- The name of the table to delete.
#
#  Returns:
#       0 - If successful.
#       1 - If it fails.
###############################################################################
function dynamodb_delete_table() {
  local table_name response
  local option OPTARG # Required to use getopts command in a function.

  # bashsupport disable=BP5008
  function usage() {
    echo "function dynamodb_delete_table"
    echo "Deletes an Amazon DynamoDB table."
    echo " -n table_name  -- The name of the table to delete."
    echo ""
  }

  # Retrieve the calling parameters.
  while getopts "n:h" option; do
    case "${option}" in
      n) table_name="${OPTARG}" ;;
      h)
        usage
        return 0
        ;;
      \?)
        echo "Invalid parameter"
        usage
        return 1
        ;;
    esac
  done
  export OPTIND=1

  if [[ -z "$table_name" ]]; then
    errecho "ERROR: You must provide a table name with the -n parameter."
```

```
    usage
    return 1
  fi

  iecho "Parameters:\n"
  iecho "    table_name:   $table_name"
  iecho ""

  response=$(aws dynamodb delete-table \
    --table-name "$table_name")

  local error_code=${?}

  if [[ $error_code -ne 0 ]]; then
    aws_cli_error_log $error_code
    errecho "ERROR: AWS reports delete-table operation failed.$response"
    return 1
  fi

  return 0
}
```

The utility functions used in this example.

```
###############################################################################
# function iecho
#
# This function enables the script to display the specified text only if
# the global variable $VERBOSE is set to true.
###############################################################################
function iecho() {
  if [[ $VERBOSE == true ]]; then
    echo "$@"
  fi
}

###############################################################################
# function errecho
#
# This function outputs everything sent to it to STDERR (standard error output).
###############################################################################
function errecho() {
```

```
  printf "%s\n" "$*" 1>&2
}

###############################################################################
# function aws_cli_error_log()
#
# This function is used to log the error messages from the AWS CLI.
#
# See https://docs.aws.amazon.com/cli/latest/topic/return-codes.html#cli-aws-
help-return-codes.
#
# The function expects the following argument:
#         $1 - The error code returned by the AWS CLI.
#
#  Returns:
#          0: - Success.
#
###############################################################################
function aws_cli_error_log() {
  local err_code=$1
  errecho "Error code : $err_code"
  if [ "$err_code" == 1 ]; then
    errecho "  One or more S3 transfers failed."
  elif [ "$err_code" == 2 ]; then
    errecho "  Command line failed to parse."
  elif [ "$err_code" == 130 ]; then
    errecho "  Process received SIGINT."
  elif [ "$err_code" == 252 ]; then
    errecho "  Command syntax invalid."
  elif [ "$err_code" == 253 ]; then
    errecho "  The system environment or configuration was invalid."
  elif [ "$err_code" == 254 ]; then
    errecho "  The service returned an error."
  elif [ "$err_code" == 255 ]; then
    errecho "  255 is a catch-all error."
  fi

  return 0
}
```

- For API details, see DeleteTable in *AWS CLI Command Reference*.

C++

SDK for C++

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
//! Delete an Amazon DynamoDB table.
/*!
  \sa deleteTable()
  \param tableName: The DynamoDB table name.
  \param clientConfiguration: AWS client configuration.
  \return bool: Function succeeded.
*/
bool AwsDoc::DynamoDB::deleteTable(const Aws::String &tableName,
                                   const Aws::Client::ClientConfiguration
 &clientConfiguration) {
    Aws::DynamoDB::DynamoDBClient dynamoClient(clientConfiguration);

    Aws::DynamoDB::Model::DeleteTableRequest request;
    request.SetTableName(tableName);

    const Aws::DynamoDB::Model::DeleteTableOutcome &result =
 dynamoClient.DeleteTable(
            request);
    if (result.IsSuccess()) {
        std::cout << "Your table \""
                  << result.GetResult().GetTableDescription().GetTableName()
                  << " was deleted.\n";
    }
    else {
        std::cerr << "Failed to delete table: " << result.GetError().GetMessage()
                  << std::endl;
    }

    return result.IsSuccess();
}
```

- For API details, see DeleteTable in *AWS SDK for C++ API Reference*.

CLI

AWS CLI

To delete a table

The following `delete-table` example deletes the `MusicCollection` table.

```
aws dynamodb delete-table \
    --table-name MusicCollection
```

Output:

```
{
    "TableDescription": {
        "TableStatus": "DELETING",
        "TableSizeBytes": 0,
        "ItemCount": 0,
        "TableName": "MusicCollection",
        "ProvisionedThroughput": {
            "NumberOfDecreasesToday": 0,
            "WriteCapacityUnits": 5,
            "ReadCapacityUnits": 5
        }
    }
}
```

For more information, see Deleting a Table in the *Amazon DynamoDB Developer Guide*.

- For API details, see DeleteTable in *AWS CLI Command Reference*.

Go

SDK for Go V2

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import (
 "context"
 "errors"
 "log"
 "time"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/attributevalue"
 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/expression"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb/types"
)

// TableBasics encapsulates the Amazon DynamoDB service actions used in the
 examples.
// It contains a DynamoDB service client that is used to act on the specified
 table.
type TableBasics struct {
 DynamoDbClient *dynamodb.Client
 TableName      string
}



// DeleteTable deletes the DynamoDB table and all of its data.
func (basics TableBasics) DeleteTable(ctx context.Context) error {
 _, err := basics.DynamoDbClient.DeleteTable(ctx, &dynamodb.DeleteTableInput{
  TableName: aws.String(basics.TableName)})
 if err != nil {
  log.Printf("Couldn't delete table %v. Here's why: %v\n", basics.TableName, err)
 }
 return err
}
```

- For API details, see DeleteTable in *AWS SDK for Go API Reference*.

Java

SDK for Java 2.x

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.DeleteTableRequest;

/**
 * Before running this Java V2 code example, set up your development
 * environment, including your credentials.
 *
 * For more information, see the following documentation topic:
 *
 * https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-
started.html
 */

public class DeleteTable {
    public static void main(String[] args) {
        final String usage = """

                Usage:
                    <tableName>

                Where:
                    tableName - The Amazon DynamoDB table to delete (for example,
 Music3).

                **Warning** This program will delete the table that you specify!
                """;

        if (args.length != 1) {
            System.out.println(usage);
            System.exit(1);
```

```
        }

        String tableName = args[0];
        System.out.format("Deleting the Amazon DynamoDB table %s...\n",
 tableName);
        Region region = Region.US_EAST_1;
        DynamoDbClient ddb = DynamoDbClient.builder()
                .region(region)
                .build();

        deleteDynamoDBTable(ddb, tableName);
        ddb.close();
    }

    public static void deleteDynamoDBTable(DynamoDbClient ddb, String tableName)
 {
        DeleteTableRequest request = DeleteTableRequest.builder()
                .tableName(tableName)
                .build();

        try {
            ddb.deleteTable(request);

        } catch (DynamoDbException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
        System.out.println(tableName + " was successfully deleted!");
    }
}
```

- For API details, see DeleteTable in *AWS SDK for Java 2.x API Reference*.

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import { DeleteTableCommand, DynamoDBClient } from "@aws-sdk/client-dynamodb";

const client = new DynamoDBClient({});

export const main = async () => {
  const command = new DeleteTableCommand({
    TableName: "DecafCoffees",
  });

  const response = await client.send(command);
  console.log(response);
  return response;
};
```

- For API details, see DeleteTable in *AWS SDK for JavaScript API Reference*.

SDK for JavaScript (v2)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create the DynamoDB service object
var ddb = new AWS.DynamoDB({ apiVersion: "2012-08-10" });

var params = {
  TableName: process.argv[2],
};

// Call DynamoDB to delete the specified table
ddb.deleteTable(params, function (err, data) {
  if (err && err.code === "ResourceNotFoundException") {
    console.log("Error: Table not found");
  } else if (err && err.code === "ResourceInUseException") {
```

```
    console.log("Error: Table in use");
  } else {
    console.log("Success", data);
  }
});
```

- For more information, see AWS SDK for JavaScript Developer Guide.
- For API details, see DeleteTable in *AWS SDK for JavaScript API Reference*.

Kotlin

SDK for Kotlin

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
suspend fun deleteDynamoDBTable(tableNameVal: String) {
    val request =
        DeleteTableRequest {
            tableName = tableNameVal
        }

    DynamoDbClient.fromEnvironment { region = "us-east-1" }.use { ddb ->
        ddb.deleteTable(request)
        println("$tableNameVal was deleted")
    }
}
```

- For API details, see DeleteTable in *AWS SDK for Kotlin API reference*.

PHP

SDK for PHP

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
public function deleteTable(string $TableName)
{
    $this->customWaiter(function () use ($TableName) {
        return $this->dynamoDbClient->deleteTable([
            'TableName' => $TableName,
        ]);
    });
}
```

- For API details, see DeleteTable in *AWS SDK for PHP API Reference*.

PowerShell

Tools for PowerShell V4

Example 1: Deletes the specified table. You are prompted for confirmation before the operation proceeds.

```
Remove-DDBTable -TableName "myTable"
```

Example 2: Deletes the specified table. You are not prompted for confirmation before the operation proceeds.

```
Remove-DDBTable -TableName "myTable" -Force
```

- For API details, see DeleteTable in *AWS Tools for PowerShell Cmdlet Reference (V4)*.

Tools for PowerShell V5

Example 1: Deletes the specified table. You are prompted for confirmation before the operation proceeds.

```
Remove-DDBTable -TableName "myTable"
```

Example 2: Deletes the specified table. You are not prompted for confirmation before the operation proceeds.

```
Remove-DDBTable -TableName "myTable" -Force
```

- For API details, see DeleteTable in *AWS Tools for PowerShell Cmdlet Reference (V5)*.

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
class Movies:
    """Encapsulates an Amazon DynamoDB table of movie data.

    Example data structure for a movie record in this table:
        {
            "year": 1999,
            "title": "For Love of the Game",
            "info": {
                "directors": ["Sam Raimi"],
                "release_date": "1999-09-15T00:00:00Z",
                "rating": 6.3,
                "plot": "A washed up pitcher flashes through his career.",
                "rank": 4987,
                "running_time_secs": 8220,
                "actors": [
```

```
                "Kevin Costner",
                "Kelly Preston",
                "John C. Reilly"
            ]
        }
    }
"""

def __init__(self, dyn_resource):
    """
    :param dyn_resource: A Boto3 DynamoDB resource.
    """
    self.dyn_resource = dyn_resource
    # The table variable is set during the scenario in the call to
    # 'exists' if the table exists. Otherwise, it is set by 'create_table'.
    self.table = None


def delete_table(self):
    """
    Deletes the table.
    """
    try:
        self.table.delete()
        self.table = None
    except ClientError as err:
        logger.error(
            "Couldn't delete table. Here's why: %s: %s",
            err.response["Error"]["Code"],
            err.response["Error"]["Message"],
        )
        raise
```

- For API details, see DeleteTable in *AWS SDK for Python (Boto3) API Reference*.

Ruby

SDK for Ruby

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
# Encapsulates an Amazon DynamoDB table of movie data.
class Scaffold
  attr_reader :dynamo_resource, :table_name, :table

  def initialize(table_name)
    client = Aws::DynamoDB::Client.new(region: 'us-east-1')
    @dynamo_resource = Aws::DynamoDB::Resource.new(client: client)
    @table_name = table_name
    @table = nil
    @logger = Logger.new($stdout)
    @logger.level = Logger::DEBUG
  end

  # Deletes the table.
  def delete_table
    @table.delete
    @table = nil
  rescue Aws::DynamoDB::Errors::ServiceError => e
    puts("Couldn't delete table. Here's why:")
    puts("\t#{e.code}: #{e.message}")
    raise
  end
```

- For API details, see DeleteTable in *AWS SDK for Ruby API Reference*.

Rust

SDK for Rust

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
pub async fn delete_table(client: &Client, table: &str) ->
 Result<DeleteTableOutput, Error> {
    let resp = client.delete_table().table_name(table).send().await;

    match resp {
        Ok(out) => {
            println!("Deleted table");
            Ok(out)
        }
        Err(e) => Err(Error::Unhandled(e.into())),
    }
}
```

- For API details, see DeleteTable in *AWS SDK for Rust API reference*.

SAP ABAP

SDK for SAP ABAP

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    TRY.
        lo_dyn->deletetable( iv_tablename = iv_table_name ).
        " Wait till the table is actually deleted.
        lo_dyn->get_waiter( )->tablenotexists(
```

```
          iv_max_wait_time = 200
          iv_tablename     = iv_table_name ).
        MESSAGE 'Table ' && iv_table_name && ' deleted.' TYPE 'I'.
      CATCH /aws1/cx_dynresourcenotfoundex.
        MESSAGE 'The table ' && iv_table_name && ' does not exist' TYPE 'E'.
      CATCH /aws1/cx_dynresourceinuseex.
        MESSAGE 'The table cannot be deleted since it is in use' TYPE 'E'.
    ENDTRY.
```

- For API details, see DeleteTable in *AWS SDK for SAP ABAP API reference*.

Swift

SDK for Swift

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import AWSDynamoDB


    ///
    /// Deletes the table from Amazon DynamoDB.
    ///
    func deleteTable() async throws {
        do {
            guard let client = self.ddbClient else {
                throw MoviesError.UninitializedClient
            }

            let input = DeleteTableInput(
                tableName: self.tableName
            )
            _ = try await client.deleteTable(input: input)
        } catch {
            print("ERROR: deleteTable:", dump(error))
            throw error
        }
```

```
}
```

- For API details, see DeleteTable in *AWS SDK for Swift API reference*.

For a complete list of AWS SDK developer guides and code examples, see Using DynamoDB with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use `DescribeTable` with an AWS SDK or CLI

The following code examples show how to use `DescribeTable`.

Action examples are code excerpts from larger programs and must be run in context. You can see this action in context in the following code examples:

- Learn the basics
- Create and manage global tables demonstrating MREC
- Create and manage MRSC global tables
- Work with global tables and multi-Region replication eventual consistency (MREC)
- Work with Streams and Time-to-Live
- Work with table encryption

.NET

SDK for .NET

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
private static async Task GetTableInformation()
{
    Console.WriteLine("\n*** Retrieving table information ***");
```

```
        var response = await Client.DescribeTableAsync(new DescribeTableRequest
        {
            TableName = ExampleTableName
        });

        var table = response.Table;
        Console.WriteLine($"Name: {table.TableName}");
        Console.WriteLine($"# of items: {table.ItemCount}");

    }
```

- For API details, see DescribeTable in *AWS SDK for .NET API Reference*.

Bash

AWS CLI with Bash script

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
###############################################################################
# function dynamodb_describe_table
#
# This function returns the status of a DynamoDB table.
#
# Parameters:
#       -n table_name  -- The name of the table.
#
#  Response:
#       - TableStatus:
#     And:
#       0 - Table is active.
#       1 - If it fails.
###############################################################################
function dynamodb_describe_table {
  local table_name
  local option OPTARG # Required to use getopts command in a function.
```

```
#######################################
# Function usage explanation
#######################################
function usage() {
  echo "function dynamodb_describe_table"
  echo "Describe the status of a DynamoDB table."
  echo "  -n table_name  -- The name of the table."
  echo ""
}

# Retrieve the calling parameters.
while getopts "n:h" option; do
  case "${option}" in
    n) table_name="${OPTARG}" ;;
    h)
      usage
      return 0
      ;;
    \?)
      echo "Invalid parameter"
      usage
      return 1
      ;;
  esac
done
export OPTIND=1

if [[ -z "$table_name" ]]; then
  errecho "ERROR: You must provide a table name with the -n parameter."
  usage
  return 1
fi

local table_status
  table_status=$(
    aws dynamodb describe-table \
      --table-name "$table_name" \
      --output text \
      --query 'Table.TableStatus'
  )

 local error_code=${?}

  if [[ $error_code -ne 0 ]]; then
```

```
      aws_cli_error_log "$error_code"
      errecho "ERROR: AWS reports describe-table operation failed.$table_status"
      return 1
    fi

  echo "$table_status"

  return 0
}
```

The utility functions used in this example.

```
###############################################################################
# function errecho
#
# This function outputs everything sent to it to STDERR (standard error output).
###############################################################################
function errecho() {
  printf "%s\n" "$*" 1>&2
}

###############################################################################
# function aws_cli_error_log()
#
# This function is used to log the error messages from the AWS CLI.
#
# See https://docs.aws.amazon.com/cli/latest/topic/return-codes.html#cli-aws-
help-return-codes.
#
# The function expects the following argument:
#         $1 - The error code returned by the AWS CLI.
#
#  Returns:
#          0: - Success.
#
###############################################################################
function aws_cli_error_log() {
  local err_code=$1
  errecho "Error code : $err_code"
  if [ "$err_code" == 1 ]; then
    errecho "  One or more S3 transfers failed."
  elif [ "$err_code" == 2 ]; then
```

```
    errecho "  Command line failed to parse."
  elif [ "$err_code" == 130 ]; then
    errecho "  Process received SIGINT."
  elif [ "$err_code" == 252 ]; then
    errecho "  Command syntax invalid."
  elif [ "$err_code" == 253 ]; then
    errecho "  The system environment or configuration was invalid."
  elif [ "$err_code" == 254 ]; then
    errecho "  The service returned an error."
  elif [ "$err_code" == 255 ]; then
    errecho "  255 is a catch-all error."
  fi

  return 0
}
```

- For API details, see DescribeTable in *AWS CLI Command Reference*.

C++

SDK for C++

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
//! Describe an Amazon DynamoDB table.
/*!
  \sa describeTable()
  \param tableName: The DynamoDB table name.
  \param clientConfiguration: AWS client configuration.
  \return bool: Function succeeded.
*/
bool AwsDoc::DynamoDB::describeTable(const Aws::String &tableName,
                                     const Aws::Client::ClientConfiguration
 &clientConfiguration) {
    Aws::DynamoDB::DynamoDBClient dynamoClient(clientConfiguration);

    Aws::DynamoDB::Model::DescribeTableRequest request;
```

```
    request.SetTableName(tableName);

    const Aws::DynamoDB::Model::DescribeTableOutcome &outcome =
 dynamoClient.DescribeTable(
            request);

    if (outcome.IsSuccess()) {
        const Aws::DynamoDB::Model::TableDescription &td =
 outcome.GetResult().GetTable();
        std::cout << "Table name  : " << td.GetTableName() << std::endl;
        std::cout << "Table ARN   : " << td.GetTableArn() << std::endl;
        std::cout << "Status      : "
                  <<
 Aws::DynamoDB::Model::TableStatusMapper::GetNameForTableStatus(
                       td.GetTableStatus()) << std::endl;
        std::cout << "Item count  : " << td.GetItemCount() << std::endl;
        std::cout << "Size (bytes): " << td.GetTableSizeBytes() << std::endl;

        const Aws::DynamoDB::Model::ProvisionedThroughputDescription &ptd =
 td.GetProvisionedThroughput();
        std::cout << "Throughput" << std::endl;
        std::cout << "  Read Capacity : " << ptd.GetReadCapacityUnits() <<
 std::endl;
        std::cout << "  Write Capacity: " << ptd.GetWriteCapacityUnits() <<
 std::endl;

        const Aws::Vector<Aws::DynamoDB::Model::AttributeDefinition> &ad =
 td.GetAttributeDefinitions();
        std::cout << "Attributes" << std::endl;
        for (const auto &a: ad)
            std::cout << "  " << a.GetAttributeName() << " (" <<

 Aws::DynamoDB::Model::ScalarAttributeTypeMapper::GetNameForScalarAttributeType(
                             a.GetAttributeType()) <<
                     ")" << std::endl;
    }
    else {
        std::cerr << "Failed to describe table: " <<
 outcome.GetError().GetMessage();
    }

    return outcome.IsSuccess();
}
```

- For API details, see DescribeTable in *AWS SDK for C++ API Reference*.

CLI

AWS CLI

To describe a table

The following `describe-table` example describes the `MusicCollection` table.

```
aws dynamodb describe-table \
    --table-name MusicCollection
```

Output:

```
{
    "Table": {
        "AttributeDefinitions": [
            {
                "AttributeName": "Artist",
                "AttributeType": "S"
            },
            {
                "AttributeName": "SongTitle",
                "AttributeType": "S"
            }
        ],
        "ProvisionedThroughput": {
            "NumberOfDecreasesToday": 0,
            "WriteCapacityUnits": 5,
            "ReadCapacityUnits": 5
        },
        "TableSizeBytes": 0,
        "TableName": "MusicCollection",
        "TableStatus": "ACTIVE",
        "KeySchema": [
            {
                "KeyType": "HASH",
                "AttributeName": "Artist"
            },
            {
```

```
                "KeyType": "RANGE",
                "AttributeName": "SongTitle"
            }
        ],
        "ItemCount": 0,
        "CreationDateTime": 1421866952.062
    }
}
```

For more information, see Describing a Table in the *Amazon DynamoDB Developer Guide*.

- For API details, see DescribeTable in *AWS CLI Command Reference*.

Go

SDK for Go V2

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import (
 "context"
 "errors"
 "log"
 "time"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/attributevalue"
 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/expression"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb/types"
)

// TableBasics encapsulates the Amazon DynamoDB service actions used in the
 examples.
// It contains a DynamoDB service client that is used to act on the specified
 table.
type TableBasics struct {
```

```
 DynamoDbClient *dynamodb.Client
 TableName      string
}



// TableExists determines whether a DynamoDB table exists.
func (basics TableBasics) TableExists(ctx context.Context) (bool, error) {
 exists := true
 _, err := basics.DynamoDbClient.DescribeTable(
  ctx, &dynamodb.DescribeTableInput{TableName: aws.String(basics.TableName)},
 )
 if err != nil {
  var notFoundEx *types.ResourceNotFoundException
  if errors.As(err, &notFoundEx) {
   log.Printf("Table %v does not exist.\n", basics.TableName)
   err = nil
  } else {
   log.Printf("Couldn't determine existence of table %v. Here's why: %v\n",
 basics.TableName, err)
  }
  exists = false
 }
 return exists, err
}
```

- For API details, see DescribeTable in *AWS SDK for Go API Reference*.

Java

SDK for Java 2.x

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
```

```
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeDefinition;
import software.amazon.awssdk.services.dynamodb.model.DescribeTableRequest;
import
 software.amazon.awssdk.services.dynamodb.model.ProvisionedThroughputDescription;
import software.amazon.awssdk.services.dynamodb.model.TableDescription;
import java.util.List;

/**
 * Before running this Java V2 code example, set up your development
 * environment, including your credentials.
 *
 * For more information, see the following documentation topic:
 *
 * https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-
started.html
 */
public class DescribeTable {
    public static void main(String[] args) {
        final String usage = """

                Usage:
                    <tableName>

                Where:
                    tableName - The Amazon DynamoDB table to get information
 about (for example, Music3).
                """;

        if (args.length != 1) {
            System.out.println(usage);
            System.exit(1);
        }

        String tableName = args[0];
        System.out.format("Getting description for %s\n\n", tableName);
        Region region = Region.US_EAST_1;
        DynamoDbClient ddb = DynamoDbClient.builder()
                .region(region)
                .build();

        describeDymamoDBTable(ddb, tableName);
        ddb.close();
    }
```

```
   public static void describeDymamoDBTable(DynamoDbClient ddb, String
tableName) {
       DescribeTableRequest request = DescribeTableRequest.builder()
               .tableName(tableName)
               .build();

       try {
           TableDescription tableInfo = ddb.describeTable(request).table();
           if (tableInfo != null) {
               System.out.format("Table name  : %s\n", tableInfo.tableName());
               System.out.format("Table ARN   : %s\n", tableInfo.tableArn());
               System.out.format("Status      : %s\n", tableInfo.tableStatus());
               System.out.format("Item count  : %d\n", tableInfo.itemCount());
               System.out.format("Size (bytes): %d\n",
tableInfo.tableSizeBytes());

               ProvisionedThroughputDescription throughputInfo =
tableInfo.provisionedThroughput();
               System.out.println("Throughput");
               System.out.format("  Read Capacity : %d\n",
throughputInfo.readCapacityUnits());
               System.out.format("  Write Capacity: %d\n",
throughputInfo.writeCapacityUnits());

               List<AttributeDefinition> attributes =
tableInfo.attributeDefinitions();
               System.out.println("Attributes");
               for (AttributeDefinition a : attributes) {
                   System.out.format("  %s (%s)\n", a.attributeName(),
a.attributeType());
               }
           }

       } catch (DynamoDbException e) {
           System.err.println(e.getMessage());
           System.exit(1);
       }
       System.out.println("\nDone!");
   }
}
```

- For API details, see DescribeTable in *AWS SDK for Java 2.x API Reference*.

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import { DescribeTableCommand, DynamoDBClient } from "@aws-sdk/client-dynamodb";

const client = new DynamoDBClient({});

export const main = async () => {
  const command = new DescribeTableCommand({
    TableName: "Pastries",
  });

  const response = await client.send(command);
  console.log(`TABLE NAME: ${response.Table.TableName}`);
  console.log(`TABLE ITEM COUNT: ${response.Table.ItemCount}`);
  return response;
};
```

- For more information, see AWS SDK for JavaScript Developer Guide.
- For API details, see DescribeTable in *AWS SDK for JavaScript API Reference*.

SDK for JavaScript (v2)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
```

```
AWS.config.update({ region: "REGION" });

// Create the DynamoDB service object
var ddb = new AWS.DynamoDB({ apiVersion: "2012-08-10" });

var params = {
  TableName: process.argv[2],
};

// Call DynamoDB to retrieve the selected table descriptions
ddb.describeTable(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data.Table.KeySchema);
  }
});
```

- For more information, see AWS SDK for JavaScript Developer Guide.
- For API details, see DescribeTable in *AWS SDK for JavaScript API Reference*.

PowerShell

Tools for PowerShell V4

Example 1: Returns details of the specified table.

```
Get-DDBTable -TableName "myTable"
```

- For API details, see DescribeTable in *AWS Tools for PowerShell Cmdlet Reference (V4)*.

Tools for PowerShell V5

Example 1: Returns details of the specified table.

```
Get-DDBTable -TableName "myTable"
```

- For API details, see DescribeTable in *AWS Tools for PowerShell Cmdlet Reference (V5)*.

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
class Movies:
    """Encapsulates an Amazon DynamoDB table of movie data.

    Example data structure for a movie record in this table:
        {
            "year": 1999,
            "title": "For Love of the Game",
            "info": {
                "directors": ["Sam Raimi"],
                "release_date": "1999-09-15T00:00:00Z",
                "rating": 6.3,
                "plot": "A washed up pitcher flashes through his career.",
                "rank": 4987,
                "running_time_secs": 8220,
                "actors": [
                    "Kevin Costner",
                    "Kelly Preston",
                    "John C. Reilly"
                ]
            }
        }
    """

    def __init__(self, dyn_resource):
        """
        :param dyn_resource: A Boto3 DynamoDB resource.
        """
        self.dyn_resource = dyn_resource
        # The table variable is set during the scenario in the call to
        # 'exists' if the table exists. Otherwise, it is set by 'create_table'.
        self.table = None
```

```
    def exists(self, table_name):
        """
        Determines whether a table exists. As a side effect, stores the table in
        a member variable.

        :param table_name: The name of the table to check.
        :return: True when the table exists; otherwise, False.
        """
        try:
            table = self.dyn_resource.Table(table_name)
            table.load()
            exists = True
        except ClientError as err:
            if err.response["Error"]["Code"] == "ResourceNotFoundException":
                exists = False
            else:
                logger.error(
                    "Couldn't check for existence of %s. Here's why: %s: %s",
                    table_name,
                    err.response["Error"]["Code"],
                    err.response["Error"]["Message"],
                )
                raise
        else:
            self.table = table
        return exists
```

- For API details, see DescribeTable in *AWS SDK for Python (Boto3) API Reference*.

Ruby

SDK for Ruby

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
# Encapsulates an Amazon DynamoDB table of movie data.
class Scaffold
  attr_reader :dynamo_resource, :table_name, :table

  def initialize(table_name)
    client = Aws::DynamoDB::Client.new(region: 'us-east-1')
    @dynamo_resource = Aws::DynamoDB::Resource.new(client: client)
    @table_name = table_name
    @table = nil
    @logger = Logger.new($stdout)
    @logger.level = Logger::DEBUG
  end

  # Determines whether a table exists. As a side effect, stores the table in
  # a member variable.
  #
  # @param table_name [String] The name of the table to check.
  # @return [Boolean] True when the table exists; otherwise, False.
  def exists?(table_name)
    @dynamo_resource.client.describe_table(table_name: table_name)
    @logger.debug("Table #{table_name} exists")
  rescue Aws::DynamoDB::Errors::ResourceNotFoundException
    @logger.debug("Table #{table_name} doesn't exist")
    false
  rescue Aws::DynamoDB::Errors::ServiceError => e
    puts("Couldn't check for existence of #{table_name}:\n")
    puts("\t#{e.code}: #{e.message}")
    raise
  end
```

- For API details, see DescribeTable in *AWS SDK for Ruby API Reference*.

SAP ABAP

SDK for SAP ABAP

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    TRY.
        oo_result = lo_dyn->describetable( iv_tablename = iv_table_name ).
        DATA(lv_tablename) = oo_result->get_table( )->ask_tablename( ).
        DATA(lv_tablearn) = oo_result->get_table( )->ask_tablearn( ).
        DATA(lv_tablestatus) = oo_result->get_table( )->ask_tablestatus( ).
        DATA(lv_itemcount) = oo_result->get_table( )->ask_itemcount( ).
        MESSAGE 'The table name is ' && lv_tablename
            && '. The table ARN is ' && lv_tablearn
            && '. The tablestatus is ' && lv_tablestatus
            && '. Item count is ' && lv_itemcount TYPE 'I'.
      CATCH /aws1/cx_dynresourcenotfoundex.
        MESSAGE 'The table ' && lv_tablename && ' does not exist' TYPE 'E'.
    ENDTRY.
```

- For API details, see DescribeTable in *AWS SDK for SAP ABAP API reference*.

For a complete list of AWS SDK developer guides and code examples, see Using DynamoDB with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use `DescribeTimeToLive` with an AWS SDK or CLI

The following code examples show how to use `DescribeTimeToLive`.

Action examples are code excerpts from larger programs and must be run in context. You can see this action in context in the following code example:

- Work with Streams and Time-to-Live

CLI

AWS CLI

To view Time to Live settings for a table

The following `describe-time-to-live` example displays Time to Live settings for the `MusicCollection` table.

```
aws dynamodb describe-time-to-live \
```

```
    --table-name MusicCollection
```

Output:

```
{
    "TimeToLiveDescription": {
        "TimeToLiveStatus": "ENABLED",
        "AttributeName": "ttl"
    }
}
```

For more information, see Time to Live in the *Amazon DynamoDB Developer Guide*.

- For API details, see DescribeTimeToLive in *AWS CLI Command Reference*.

Java

SDK for Java 2.x

Describe TTL configuration on an existing DynamoDB table using AWS SDK for Java 2.x.

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.DescribeTimeToLiveRequest;
import software.amazon.awssdk.services.dynamodb.model.DescribeTimeToLiveResponse;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.ResourceNotFoundException;

import java.util.logging.Level;
import java.util.logging.Logger;

    public DescribeTimeToLiveResponse describeTTL(final String tableName, final
 Region region) {
        final DescribeTimeToLiveRequest request =
            DescribeTimeToLiveRequest.builder().tableName(tableName).build();

        try (DynamoDbClient ddb = dynamoDbClient != null
            ? dynamoDbClient
            : DynamoDbClient.builder().region(region).build()) {
            return ddb.describeTimeToLive(request);
        } catch (ResourceNotFoundException e) {
            System.err.format(TABLE_NOT_FOUND_ERROR, tableName);
```

```
            throw e;
        } catch (DynamoDbException e) {
            System.err.println(e.getMessage());
            throw e;
        }
    }
```

- For API details, see DescribeTimeToLive in *AWS SDK for Java 2.x API Reference*.

JavaScript

SDK for JavaScript (v3)

Describe TTL configuration on an existing DynamoDB table using AWS SDK for JavaScript.

```
import { DynamoDBClient, DescribeTimeToLiveCommand } from "@aws-sdk/client-
dynamodb";

export const describeTTL = async (tableName, region) => {
    const client = new DynamoDBClient({
        region: region,
        endpoint: `https://dynamodb.${region}.amazonaws.com`
    });

    try {
        const ttlDescription = await client.send(new
 DescribeTimeToLiveCommand({ TableName: tableName }));

        if (ttlDescription.TimeToLiveDescription.TimeToLiveStatus === 'ENABLED')
 {
            console.log("TTL is enabled for table %s.", tableName);
        } else {
            console.log("TTL is not enabled for table %s.", tableName);
        }

        return ttlDescription;
    } catch (e) {
        console.error(`Error describing table: ${e}`);
        throw e;
    }
}
```

```
// Example usage (commented out for testing)
// describeTTL('your-table-name', 'us-east-1');
```

- For API details, see DescribeTimeToLive in *AWS SDK for JavaScript API Reference*.

Python

SDK for Python (Boto3)

Describe TTL configuration on an existing DynamoDB table using AWS SDK for Python (Boto3).

```
import boto3


def describe_ttl(table_name, region):
    """
    Describes TTL on an existing table, as well as a region.

    :param table_name: String representing the name of the table
    :param region: AWS Region of the table - example `us-east-1`
    :return: Time to live description.
    """
    try:
        dynamodb = boto3.resource("dynamodb", region_name=region)
        ttl_description = dynamodb.describe_time_to_live(TableName=table_name)
        print(
            f"TimeToLive for table {table_name} is status
 {ttl_description['TimeToLiveDescription']['TimeToLiveStatus']}"
        )

        return ttl_description
    except Exception as e:
        print(f"Error describing table: {e}")
        raise


# Enter your own table name and AWS region
describe_ttl("your-table-name", "us-east-1")
```

- For API details, see DescribeTimeToLive in *AWS SDK for Python (Boto3) API Reference*.

For a complete list of AWS SDK developer guides and code examples, see Using DynamoDB with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use `ExecuteStatement` with an AWS SDK

The following code examples show how to use `ExecuteStatement`.

Action examples are code excerpts from larger programs and must be run in context. You can see this action in context in the following code examples:

- Delete data using PartiQL DELETE
- Insert data using PartiQL INSERT
- Query a table using PartiQL
- Query data using PartiQL SELECT
- Update data using PartiQL UPDATE

.NET

SDK for .NET

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Use an INSERT statement to add an item.

```
        /// <summary>
        /// Inserts a single movie into the movies table.
        /// </summary>
        /// <param name="tableName">The name of the table.</param>
        /// <param name="movieTitle">The title of the movie to insert.</param>
        /// <param name="year">The year that the movie was released.</param>
        /// <returns>A Boolean value that indicates the success or failure of
        /// the INSERT operation.</returns>
        public static async Task<bool> InsertSingleMovie(string tableName, string
movieTitle, int year)
```

```
        {
            string insertBatch = $"INSERT INTO {tableName} VALUE {{'title': ?,
'year': ?}}";

            var response = await Client.ExecuteStatementAsync(new
ExecuteStatementRequest
            {
                Statement = insertBatch,
                Parameters = new List<AttributeValue>
                {
                    new AttributeValue { S = movieTitle },
                    new AttributeValue { N = year.ToString() },
                },
            });

            return response.HttpStatusCode == System.Net.HttpStatusCode.OK;
        }
```

Use a SELECT statement to get an item.

```
        /// <summary>
        /// Uses a PartiQL SELECT statement to retrieve a single movie from the
        /// movie database.
        /// </summary>
        /// <param name="tableName">The name of the movie table.</param>
        /// <param name="movieTitle">The title of the movie to retrieve.</param>
        /// <returns>A list of movie data. If no movie matches the supplied
        /// title, the list is empty.</returns>
        public static async Task<List<Dictionary<string, AttributeValue>>>
GetSingleMovie(string tableName, string movieTitle)
        {
            string selectSingle = $"SELECT * FROM {tableName} WHERE title = ?";
            var parameters = new List<AttributeValue>
            {
                new AttributeValue { S = movieTitle },
            };

            var response = await Client.ExecuteStatementAsync(new
ExecuteStatementRequest
            {
```

```
                Statement = selectSingle,
                Parameters = parameters,
            });

            return response.Items;
        }
```

Use a SELECT statement to get a list of items.

```
        /// <summary>
        /// Retrieve multiple movies by year using a SELECT statement.
        /// </summary>
        /// <param name="tableName">The name of the movie table.</param>
        /// <param name="year">The year the movies were released.</param>
        /// <returns></returns>
        public static async Task<List<Dictionary<string, AttributeValue>>>
GetMovies(string tableName, int year)
        {
            string selectSingle = $"SELECT * FROM {tableName} WHERE year = ?";
            var parameters = new List<AttributeValue>
            {
                new AttributeValue { N = year.ToString() },
            };

            var response = await Client.ExecuteStatementAsync(new
ExecuteStatementRequest
            {
                Statement = selectSingle,
                Parameters = parameters,
            });

            return response.Items;
        }
```

Use an UPDATE statement to update an item.

```
        /// <summary>
        /// Updates a single movie in the table, adding information for the
```

```
        /// producer.
        /// </summary>
        /// <param name="tableName">the name of the table.</param>
        /// <param name="producer">The name of the producer.</param>
        /// <param name="movieTitle">The movie title.</param>
        /// <param name="year">The year the movie was released.</param>
        /// <returns>A Boolean value that indicates the success of the
        /// UPDATE operation.</returns>
        public static async Task<bool> UpdateSingleMovie(string tableName, string
producer, string movieTitle, int year)
        {
            string insertSingle = $"UPDATE {tableName} SET Producer=? WHERE title
= ? AND year = ?";

            var response = await Client.ExecuteStatementAsync(new
ExecuteStatementRequest
            {
                Statement = insertSingle,
                Parameters = new List<AttributeValue>
                {
                    new AttributeValue { S = producer },
                    new AttributeValue { S = movieTitle },
                    new AttributeValue { N = year.ToString() },
                },
            });

            return response.HttpStatusCode == System.Net.HttpStatusCode.OK;
        }
```

Use a DELETE statement to delete a single movie.

```
        /// <summary>
        /// Deletes a single movie from the table.
        /// </summary>
        /// <param name="tableName">The name of the table.</param>
        /// <param name="movieTitle">The title of the movie to delete.</param>
        /// <param name="year">The year that the movie was released.</param>
        /// <returns>A Boolean value that indicates the success of the
        /// DELETE operation.</returns>
```

```
        public static async Task<bool> DeleteSingleMovie(string tableName, string
movieTitle, int year)
        {
            var deleteSingle = $"DELETE FROM {tableName} WHERE title = ? AND year
= ?";

            var response = await Client.ExecuteStatementAsync(new
ExecuteStatementRequest
            {
                Statement = deleteSingle,
                Parameters = new List<AttributeValue>
                {
                    new AttributeValue { S = movieTitle },
                    new AttributeValue { N = year.ToString() },
                },
            });

            return response.HttpStatusCode == System.Net.HttpStatusCode.OK;
        }
```

- For API details, see ExecuteStatement in *AWS SDK for .NET API Reference*.

C++

SDK for C++

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Use an INSERT statement to add an item.

```
    Aws::DynamoDB::DynamoDBClient dynamoClient(clientConfiguration);

    // 2. Add a new movie using an "Insert" statement. (ExecuteStatement)
    Aws::String title;
    float rating;
    int year;
```

```
   Aws::String plot;
   {
       title = askQuestion(
               "Enter the title of a movie you want to add to the table: ");
       year = askQuestionForInt("What year was it released? ");
       rating = askQuestionForFloatRange("On a scale of 1 - 10, how do you rate
it? ",
                                         1, 10);
       plot = askQuestion("Summarize the plot for me: ");

       Aws::DynamoDB::Model::ExecuteStatementRequest request;
       std::stringstream sqlStream;
       sqlStream << "INSERT INTO \"" << MOVIE_TABLE_NAME << "\" VALUE {'"
                 << TITLE_KEY << "': ?, '" << YEAR_KEY << "': ?, '"
                 << INFO_KEY << "': ?}";

       request.SetStatement(sqlStream.str());

       // Create the parameter attributes.
       Aws::Vector<Aws::DynamoDB::Model::AttributeValue> attributes;
       attributes.push_back(Aws::DynamoDB::Model::AttributeValue().SetS(title));
       attributes.push_back(Aws::DynamoDB::Model::AttributeValue().SetN(year));

       Aws::DynamoDB::Model::AttributeValue infoMapAttribute;

       std::shared_ptr<Aws::DynamoDB::Model::AttributeValue> ratingAttribute =
Aws::MakeShared<Aws::DynamoDB::Model::AttributeValue>(
               ALLOCATION_TAG.c_str());
       ratingAttribute->SetN(rating);
       infoMapAttribute.AddMEntry(RATING_KEY, ratingAttribute);

       std::shared_ptr<Aws::DynamoDB::Model::AttributeValue> plotAttribute =
Aws::MakeShared<Aws::DynamoDB::Model::AttributeValue>(
               ALLOCATION_TAG.c_str());
       plotAttribute->SetS(plot);
       infoMapAttribute.AddMEntry(PLOT_KEY, plotAttribute);
       attributes.push_back(infoMapAttribute);
       request.SetParameters(attributes);

       Aws::DynamoDB::Model::ExecuteStatementOutcome outcome =
dynamoClient.ExecuteStatement(
               request);

       if (!outcome.IsSuccess()) {
```

```
            std::cerr << "Failed to add a movie: " <<
outcome.GetError().GetMessage()
                      << std::endl;
            return false;
        }
    }
```

Use a SELECT statement to get an item.

```
    //  3. Get the data for the movie using a "Select" statement.
(ExecuteStatement)
    {
        Aws::DynamoDB::Model::ExecuteStatementRequest request;
        std::stringstream sqlStream;
        sqlStream << "SELECT * FROM  \"" << MOVIE_TABLE_NAME << "\" WHERE "
                  << TITLE_KEY << "=? and " << YEAR_KEY << "=?";

        request.SetStatement(sqlStream.str());

        Aws::Vector<Aws::DynamoDB::Model::AttributeValue> attributes;
        attributes.push_back(Aws::DynamoDB::Model::AttributeValue().SetS(title));
        attributes.push_back(Aws::DynamoDB::Model::AttributeValue().SetN(year));
        request.SetParameters(attributes);

        Aws::DynamoDB::Model::ExecuteStatementOutcome outcome =
dynamoClient.ExecuteStatement(
                request);

        if (!outcome.IsSuccess()) {
            std::cerr << "Failed to retrieve movie information: "
                      << outcome.GetError().GetMessage() << std::endl;
            return false;
        }
        else {
            // Print the retrieved movie information.
            const Aws::DynamoDB::Model::ExecuteStatementResult &result =
outcome.GetResult();

            const Aws::Vector<Aws::Map<Aws::String,
Aws::DynamoDB::Model::AttributeValue>> &items = result.GetItems();

            if (items.size() == 1) {
```

```
                printMovieInfo(items[0]);
            }
            else {
                std::cerr << "Error: " << items.size() << " movies were
retrieved. "
                          << " There should be only one movie." << std::endl;
            }
        }
    }
```

Use an UPDATE statement to update an item.

```
    //  4. Update the data for the movie using an "Update" statement.
(ExecuteStatement)
    {
        rating = askQuestionForFloatRange(
                Aws::String("\nLet's update your movie.\nYou rated it  ") +
                std::to_string(rating)
                + ", what new rating would you give it? ", 1, 10);

        Aws::DynamoDB::Model::ExecuteStatementRequest request;
        std::stringstream sqlStream;
        sqlStream << "UPDATE \"" << MOVIE_TABLE_NAME << "\" SET "
                  << INFO_KEY << "." << RATING_KEY << "=? WHERE "
                  << TITLE_KEY << "=? AND " << YEAR_KEY << "=?";

        request.SetStatement(sqlStream.str());

        Aws::Vector<Aws::DynamoDB::Model::AttributeValue> attributes;

attributes.push_back(Aws::DynamoDB::Model::AttributeValue().SetN(rating));
        attributes.push_back(Aws::DynamoDB::Model::AttributeValue().SetS(title));
        attributes.push_back(Aws::DynamoDB::Model::AttributeValue().SetN(year));

        request.SetParameters(attributes);

        Aws::DynamoDB::Model::ExecuteStatementOutcome outcome =
dynamoClient.ExecuteStatement(
                request);

        if (!outcome.IsSuccess()) {
            std::cerr << "Failed to update a movie: "
```

```
                    << outcome.GetError().GetMessage();
            return false;
        }
    }
```

Use a DELETE statement to delete an item.

```
    // 6. Delete the movie using a "Delete" statement. (ExecuteStatement)
    {
        Aws::DynamoDB::Model::ExecuteStatementRequest request;
        std::stringstream sqlStream;
        sqlStream << "DELETE FROM  \"" << MOVIE_TABLE_NAME << "\" WHERE "
                  << TITLE_KEY << "=? and " << YEAR_KEY << "=?";

        request.SetStatement(sqlStream.str());

        Aws::Vector<Aws::DynamoDB::Model::AttributeValue> attributes;
        attributes.push_back(Aws::DynamoDB::Model::AttributeValue().SetS(title));
        attributes.push_back(Aws::DynamoDB::Model::AttributeValue().SetN(year));
        request.SetParameters(attributes);

        Aws::DynamoDB::Model::ExecuteStatementOutcome outcome =
dynamoClient.ExecuteStatement(
                request);
        if (!outcome.IsSuccess()) {
            std::cerr << "Failed to delete the movie: "
                      << outcome.GetError().GetMessage() << std::endl;
            return false;
        }
    }
```

- For API details, see ExecuteStatement in *AWS SDK for C++ API Reference*.

Go

SDK for Go V2

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Define a function receiver struct for the example.

```
import (
 "context"
 "fmt"
 "log"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/attributevalue"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb/types"
)

// PartiQLRunner encapsulates the Amazon DynamoDB service actions used in the
// PartiQL examples. It contains a DynamoDB service client that is used to act on
 the
// specified table.
type PartiQLRunner struct {
 DynamoDbClient *dynamodb.Client
 TableName      string
}
```

Use an INSERT statement to add an item.

```
// AddMovie runs a PartiQL INSERT statement to add a movie to the DynamoDB table.
func (runner PartiQLRunner) AddMovie(ctx context.Context, movie Movie) error {
 params, err := attributevalue.MarshalList([]interface{}{movie.Title, movie.Year,
 movie.Info})
```

```
 if err != nil {
  panic(err)
 }
 _, err = runner.DynamoDbClient.ExecuteStatement(ctx,
 &dynamodb.ExecuteStatementInput{
  Statement: aws.String(
   fmt.Sprintf("INSERT INTO \"%v\" VALUE {'title': ?, 'year': ?, 'info': ?}",
    runner.TableName)),
  Parameters: params,
 })
 if err != nil {
  log.Printf("Couldn't insert an item with PartiQL. Here's why: %v\n", err)
 }
 return err
}
```

Use a SELECT statement to get an item.

```
// GetMovie runs a PartiQL SELECT statement to get a movie from the DynamoDB
 table by
// title and year.
func (runner PartiQLRunner) GetMovie(ctx context.Context, title string, year int)
 (Movie, error) {
 var movie Movie
 params, err := attributevalue.MarshalList([]interface{}{title, year})
 if err != nil {
  panic(err)
 }
 response, err := runner.DynamoDbClient.ExecuteStatement(ctx,
 &dynamodb.ExecuteStatementInput{
  Statement: aws.String(
   fmt.Sprintf("SELECT * FROM \"%v\" WHERE title=? AND year=?",
    runner.TableName)),
  Parameters: params,
 })
 if err != nil {
  log.Printf("Couldn't get info about %v. Here's why: %v\n", title, err)
 } else {
  err = attributevalue.UnmarshalMap(response.Items[0], &movie)
  if err != nil {
```

```
   log.Printf("Couldn't unmarshal response. Here's why: %v\n", err)
  }
 }
 return movie, err
}
```

Use a SELECT statement to get a list of items and project the results.

```
// GetAllMovies runs a PartiQL SELECT statement to get all movies from the
 DynamoDB table.
// pageSize is not typically required and is used to show how to paginate the
 results.
// The results are projected to return only the title and rating of each movie.
func (runner PartiQLRunner) GetAllMovies(ctx context.Context, pageSize int32)
 ([]map[string]interface{}, error) {
 var output []map[string]interface{}
 var response *dynamodb.ExecuteStatementOutput
 var err error
 var nextToken *string
 for moreData := true; moreData; {
  response, err = runner.DynamoDbClient.ExecuteStatement(ctx,
 &dynamodb.ExecuteStatementInput{
   Statement: aws.String(
    fmt.Sprintf("SELECT title, info.rating FROM \"%v\"", runner.TableName)),
   Limit:     aws.Int32(pageSize),
   NextToken: nextToken,
  })
  if err != nil {
   log.Printf("Couldn't get movies. Here's why: %v\n", err)
   moreData = false
  } else {
   var pageOutput []map[string]interface{}
   err = attributevalue.UnmarshalListOfMaps(response.Items, &pageOutput)
   if err != nil {
    log.Printf("Couldn't unmarshal response. Here's why: %v\n", err)
   } else {
    log.Printf("Got a page of length %v.\n", len(response.Items))
    output = append(output, pageOutput...)
   }
   nextToken = response.NextToken
```

```
   moreData = nextToken != nil
  }
 }
 return output, err
}
```

Use an UPDATE statement to update an item.

```
// UpdateMovie runs a PartiQL UPDATE statement to update the rating of a movie
 that
// already exists in the DynamoDB table.
func (runner PartiQLRunner) UpdateMovie(ctx context.Context, movie Movie, rating
 float64) error {
 params, err := attributevalue.MarshalList([]interface{}{rating, movie.Title,
 movie.Year})
 if err != nil {
  panic(err)
 }
 _, err = runner.DynamoDbClient.ExecuteStatement(ctx,
 &dynamodb.ExecuteStatementInput{
  Statement: aws.String(
   fmt.Sprintf("UPDATE \"%v\" SET info.rating=? WHERE title=? AND year=?",
    runner.TableName)),
  Parameters: params,
 })
 if err != nil {
  log.Printf("Couldn't update movie %v. Here's why: %v\n", movie.Title, err)
 }
 return err
}
```

Use a DELETE statement to delete an item.

```
// DeleteMovie runs a PartiQL DELETE statement to remove a movie from the
 DynamoDB table.
func (runner PartiQLRunner) DeleteMovie(ctx context.Context, movie Movie) error {
```

```
 params, err := attributevalue.MarshalList([]interface{}{movie.Title,
 movie.Year})
 if err != nil {
  panic(err)
 }
 _, err = runner.DynamoDbClient.ExecuteStatement(ctx,
 &dynamodb.ExecuteStatementInput{
  Statement: aws.String(
   fmt.Sprintf("DELETE FROM \"%v\" WHERE title=? AND year=?",
    runner.TableName)),
  Parameters: params,
 })
 if err != nil {
  log.Printf("Couldn't delete %v from the table. Here's why: %v\n", movie.Title,
 err)
 }
 return err
}
```

Define a Movie struct that is used in this example.

```
import (
 "archive/zip"
 "bytes"
 "encoding/json"
 "fmt"
 "io"
 "log"
 "net/http"

 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/attributevalue"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb/types"
)

// Movie encapsulates data about a movie. Title and Year are the composite
 primary key
// of the movie in Amazon DynamoDB. Title is the sort key, Year is the partition
 key,
// and Info is additional data.
type Movie struct {
```

```
 Title string                 `dynamodbav:"title"`
 Year  int                    `dynamodbav:"year"`
 Info  map[string]interface{} `dynamodbav:"info"`
}

// GetKey returns the composite primary key of the movie in a format that can be
// sent to DynamoDB.
func (movie Movie) GetKey() map[string]types.AttributeValue {
 title, err := attributevalue.Marshal(movie.Title)
 if err != nil {
  panic(err)
 }
 year, err := attributevalue.Marshal(movie.Year)
 if err != nil {
  panic(err)
 }
 return map[string]types.AttributeValue{"title": title, "year": year}
}

// String returns the title, year, rating, and plot of a movie, formatted for the
 example.
func (movie Movie) String() string {
 return fmt.Sprintf("%v\n\tReleased: %v\n\tRating: %v\n\tPlot: %v\n",
  movie.Title, movie.Year, movie.Info["rating"], movie.Info["plot"])
}
```

- For API details, see ExecuteStatement in *AWS SDK for Go API Reference*.

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Create an item using PartiQL.

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";

import {
  ExecuteStatementCommand,
  DynamoDBDocumentClient,
} from "@aws-sdk/lib-dynamodb";

const client = new DynamoDBClient({});
const docClient = DynamoDBDocumentClient.from(client);

export const main = async () => {
  const command = new ExecuteStatementCommand({
    Statement: `INSERT INTO Flowers value {'Name':?}`,
    Parameters: ["Rose"],
  });

  const response = await docClient.send(command);
  console.log(response);
  return response;
};
```

Get an item using PartiQL.

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";

import {
  ExecuteStatementCommand,
  DynamoDBDocumentClient,
} from "@aws-sdk/lib-dynamodb";

const client = new DynamoDBClient({});
const docClient = DynamoDBDocumentClient.from(client);

export const main = async () => {
  const command = new ExecuteStatementCommand({
    Statement: "SELECT * FROM CloudTypes WHERE IsStorm=?",
    Parameters: [false],
    ConsistentRead: true,
  });

  const response = await docClient.send(command);
  console.log(response);
```

```
  return response;
};
```

Update an item using PartiQL.

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";

import {
  ExecuteStatementCommand,
  DynamoDBDocumentClient,
} from "@aws-sdk/lib-dynamodb";

const client = new DynamoDBClient({});
const docClient = DynamoDBDocumentClient.from(client);

export const main = async () => {
  const command = new ExecuteStatementCommand({
    Statement: "UPDATE EyeColors SET IsRecessive=? where Color=?",
    Parameters: [true, "blue"],
  });

  const response = await docClient.send(command);
  console.log(response);
  return response;
};
```

Delete an item using PartiQL.

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";

import {
  ExecuteStatementCommand,
  DynamoDBDocumentClient,
} from "@aws-sdk/lib-dynamodb";

const client = new DynamoDBClient({});
const docClient = DynamoDBDocumentClient.from(client);

export const main = async () => {
  const command = new ExecuteStatementCommand({
    Statement: "DELETE FROM PaintColors where Name=?",
```

```
    Parameters: ["Purple"],
  });

  const response = await docClient.send(command);
  console.log(response);
  return response;
};
```

- For API details, see ExecuteStatement in *AWS SDK for JavaScript API Reference*.

PHP

SDK for PHP

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    public function insertItemByPartiQL(string $statement, array $parameters)
    {
        $this->dynamoDbClient->executeStatement([
            'Statement' => "$statement",
            'Parameters' => $parameters,
        ]);
    }

    public function getItemByPartiQL(string $tableName, array $key): Result
    {
        list($statement, $parameters) = $this-
>buildStatementAndParameters("SELECT", $tableName, $key['Item']);

        return $this->dynamoDbClient->executeStatement([
            'Parameters' => $parameters,
            'Statement' => $statement,
        ]);
    }

    public function updateItemByPartiQL(string $statement, array $parameters)
    {
```

```
        $this->dynamoDbClient->executeStatement([
            'Statement' => $statement,
            'Parameters' => $parameters,
        ]);
    }

    public function deleteItemByPartiQL(string $statement, array $parameters)
    {
        $this->dynamoDbClient->executeStatement([
            'Statement' => $statement,
            'Parameters' => $parameters,
        ]);
    }
```

- For API details, see ExecuteStatement in *AWS SDK for PHP API Reference*.

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
class PartiQLWrapper:
    """
    Encapsulates a DynamoDB resource to run PartiQL statements.
    """

    def __init__(self, dyn_resource):
        """
        :param dyn_resource: A Boto3 DynamoDB resource.
        """
        self.dyn_resource = dyn_resource


    def run_partiql(self, statement, params):
        """
        Runs a PartiQL statement. A Boto3 resource is used even though
```

```
        `execute_statement` is called on the underlying `client` object because
the
        resource transforms input and output from plain old Python objects
(POPOs) to
        the DynamoDB format. If you create the client directly, you must do these
        transforms yourself.

        :param statement: The PartiQL statement.
        :param params: The list of PartiQL parameters. These are applied to the
                       statement in the order they are listed.
        :return: The items returned from the statement, if any.
        """
        try:
            output = self.dyn_resource.meta.client.execute_statement(
                Statement=statement, Parameters=params
            )
        except ClientError as err:
            if err.response["Error"]["Code"] == "ResourceNotFoundException":
                logger.error(
                    "Couldn't execute PartiQL '%s' because the table does not
exist.",
                    statement,
                )
            else:
                logger.error(
                    "Couldn't execute PartiQL '%s'. Here's why: %s: %s",
                    statement,
                    err.response["Error"]["Code"],
                    err.response["Error"]["Message"],
                )
            raise
        else:
            return output
```

- For API details, see ExecuteStatement in *AWS SDK for Python (Boto3) API Reference*.

Ruby

SDK for Ruby

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Select a single item using PartiQL.

```
class DynamoDBPartiQLSingle
  attr_reader :dynamo_resource, :table

  def initialize(table_name)
    client = Aws::DynamoDB::Client.new(region: 'us-east-1')
    @dynamodb = Aws::DynamoDB::Resource.new(client: client)
    @table = @dynamodb.table(table_name)
  end

  # Gets a single record from a table using PartiQL.
  # Note: To perform more fine-grained selects,
  # use the Client.query instance method instead.
  #
  # @param title [String] The title of the movie to search.
  # @return [Aws::DynamoDB::Types::ExecuteStatementOutput]
  def select_item_by_title(title)
    request = {
      statement: "SELECT * FROM \"#{@table.name}\" WHERE title=?",
      parameters: [title]
    }
    @dynamodb.client.execute_statement(request)
  end
```

Update a single item using PartiQL.

```
class DynamoDBPartiQLSingle
  attr_reader :dynamo_resource, :table

  def initialize(table_name)
```

```
    client = Aws::DynamoDB::Client.new(region: 'us-east-1')
    @dynamodb = Aws::DynamoDB::Resource.new(client: client)
    @table = @dynamodb.table(table_name)
  end

  # Updates a single record from a table using PartiQL.
  #
  # @param title [String] The title of the movie to update.
  # @param year [Integer] The year the movie was released.
  # @param rating [Float] The new rating to assign the title.
  # @return [Aws::DynamoDB::Types::ExecuteStatementOutput]
  def update_rating_by_title(title, year, rating)
    request = {
      statement: "UPDATE \"#{@table.name}\" SET info.rating=? WHERE title=? and
 year=?",
      parameters: [{ "N": rating }, title, year]
    }
    @dynamodb.client.execute_statement(request)
  end
```

Add a single item using PartiQL.

```
class DynamoDBPartiQLSingle
  attr_reader :dynamo_resource, :table

  def initialize(table_name)
    client = Aws::DynamoDB::Client.new(region: 'us-east-1')
    @dynamodb = Aws::DynamoDB::Resource.new(client: client)
    @table = @dynamodb.table(table_name)
  end

  # Adds a single record to a table using PartiQL.
  #
  # @param title [String] The title of the movie to update.
  # @param year [Integer] The year the movie was released.
  # @param plot [String] The plot of the movie.
  # @param rating [Float] The new rating to assign the title.
  # @return [Aws::DynamoDB::Types::ExecuteStatementOutput]
  def insert_item(title, year, plot, rating)
    request = {
      statement: "INSERT INTO \"#{@table.name}\" VALUE {'title': ?, 'year': ?,
 'info': ?}",
```

```
      parameters: [title, year, { 'plot': plot, 'rating': rating }]
    }
    @dynamodb.client.execute_statement(request)
  end
```

Delete a single item using PartiQL.

```
class DynamoDBPartiQLSingle
  attr_reader :dynamo_resource, :table

  def initialize(table_name)
    client = Aws::DynamoDB::Client.new(region: 'us-east-1')
    @dynamodb = Aws::DynamoDB::Resource.new(client: client)
    @table = @dynamodb.table(table_name)
  end

  # Deletes a single record from a table using PartiQL.
  #
  # @param title [String] The title of the movie to update.
  # @param year [Integer] The year the movie was released.
  # @return [Aws::DynamoDB::Types::ExecuteStatementOutput]
  def delete_item_by_title(title, year)
    request = {
      statement: "DELETE FROM \"#{@table.name}\" WHERE title=? and year=?",
      parameters: [title, year]
    }
    @dynamodb.client.execute_statement(request)
  end
```

- For API details, see ExecuteStatement in *AWS SDK for Ruby API Reference*.

For a complete list of AWS SDK developer guides and code examples, see Using DynamoDB with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use `GetItem` with an AWS SDK or CLI

The following code examples show how to use `GetItem`.

Action examples are code excerpts from larger programs and must be run in context. You can see this action in context in the following code examples:

- Learn the basics
- Accelerate reads with DAX
- Create and manage MRSC global tables
- Work with global tables and multi-Region replication eventual consistency (MREC)

.NET

SDK for .NET (v4)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    /// <summary>
    /// Gets information about an existing movie from the table.
    /// </summary>
    /// <param name="newMovie">A Movie object containing information about
    /// the movie to retrieve.</param>
    /// <param name="tableName">The name of the table containing the movie.</
param>
    /// <returns>A Dictionary object containing information about the item
    /// retrieved.</returns>
    public async Task<Dictionary<string, AttributeValue>> GetItemAsync(Movie
 newMovie, string tableName)
    {
        try
        {
            var key = new Dictionary<string, AttributeValue>
            {
                ["title"] = new AttributeValue { S = newMovie.Title },
                ["year"] = new AttributeValue { N = newMovie.Year.ToString() },
            };

            var request = new GetItemRequest
```

```
            {
                Key = key,
                TableName = tableName,
            };

            var response = await _amazonDynamoDB.GetItemAsync(request);
            return response.Item;
        }
        catch (ResourceNotFoundException ex)
        {
            Console.WriteLine($"Table {tableName} was not found. {ex.Message}");
            return new Dictionary<string, AttributeValue>();
        }
        catch (AmazonDynamoDBException ex)
        {
            Console.WriteLine($"An Amazon DynamoDB error occurred while getting
item. {ex.Message}");
            throw;
        }
        catch (Exception ex)
        {
            Console.WriteLine($"An error occurred while getting item.
{ex.Message}");
            throw;
        }
    }
```

- For API details, see GetItem in *AWS SDK for .NET API Reference*.

Bash

AWS CLI with Bash script

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
##############################################################################
```

```
# function dynamodb_get_item
#
# This function gets an item from a DynamoDB table.
#
# Parameters:
#       -n table_name  -- The name of the table.
#       -k keys  -- Path to json file containing the keys that identify the item
 to get.
#       [-q query]  -- Optional JMESPath query expression.
#
#  Returns:
#       The item as text output.
#  And:
#       0 - If successful.
#       1 - If it fails.
###############################################################################
function dynamodb_get_item() {
  local table_name keys query response
  local option OPTARG # Required to use getopts command in a function.

  # ######################################
  # Function usage explanation
  ######################################
  function usage() {
    echo "function dynamodb_get_item"
    echo "Get an item from a DynamoDB table."
    echo " -n table_name  -- The name of the table."
    echo " -k keys  -- Path to json file containing the keys that identify the
 item to get."
    echo " [-q query]  -- Optional JMESPath query expression."
    echo ""
  }
  query=""
  while getopts "n:k:q:h" option; do
    case "${option}" in
      n) table_name="${OPTARG}" ;;
      k) keys="${OPTARG}" ;;
      q) query="${OPTARG}" ;;
      h)
        usage
        return 0
        ;;
      \?)
        echo "Invalid parameter"
```

```
      usage
      return 1
      ;;
  esac
done
export OPTIND=1

if [[ -z "$table_name" ]]; then
  errecho "ERROR: You must provide a table name with the -n parameter."
  usage
  return 1
fi

if [[ -z "$keys" ]]; then
  errecho "ERROR: You must provide a keys json file path the -k parameter."
  usage
  return 1
fi

if [[ -n "$query" ]]; then
  response=$(aws dynamodb get-item \
    --table-name "$table_name" \
    --key file://"$keys" \
    --output text \
    --query "$query")
else
  response=$(
    aws dynamodb get-item \
      --table-name "$table_name" \
      --key file://"$keys" \
      --output text
  )
fi

local error_code=${?}

if [[ $error_code -ne 0 ]]; then
  aws_cli_error_log $error_code
  errecho "ERROR: AWS reports get-item operation failed.$response"
  return 1
fi

if [[ -n "$query" ]]; then
```

```
    echo "$response" | sed "/^\t/s/\t//1" # Remove initial tab that the JMSEPath
 query inserts on some strings.
  else
    echo "$response"
  fi

  return 0
}
```

The utility functions used in this example.

```
###############################################################################
# function errecho
#
# This function outputs everything sent to it to STDERR (standard error output).
###############################################################################
function errecho() {
  printf "%s\n" "$*" 1>&2
}

###############################################################################
# function aws_cli_error_log()
#
# This function is used to log the error messages from the AWS CLI.
#
# See https://docs.aws.amazon.com/cli/latest/topic/return-codes.html#cli-aws-
help-return-codes.
#
# The function expects the following argument:
#         $1 - The error code returned by the AWS CLI.
#
#  Returns:
#          0: - Success.
#
###############################################################################
function aws_cli_error_log() {
  local err_code=$1
  errecho "Error code : $err_code"
  if [ "$err_code" == 1 ]; then
    errecho "  One or more S3 transfers failed."
  elif [ "$err_code" == 2 ]; then
    errecho "  Command line failed to parse."
```

```
  elif [ "$err_code" == 130 ]; then
    errecho "  Process received SIGINT."
  elif [ "$err_code" == 252 ]; then
    errecho "  Command syntax invalid."
  elif [ "$err_code" == 253 ]; then
    errecho "  The system environment or configuration was invalid."
  elif [ "$err_code" == 254 ]; then
    errecho "  The service returned an error."
  elif [ "$err_code" == 255 ]; then
    errecho "  255 is a catch-all error."
  fi

  return 0
}
```

- For API details, see GetItem in *AWS CLI Command Reference*.

C++

SDK for C++

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
//! Get an item from an Amazon DynamoDB table.
/*!
  \sa getItem()
  \param tableName: The table name.
  \param partitionKey: The partition key.
  \param partitionValue: The value for the partition key.
  \param clientConfiguration: AWS client configuration.
  \return bool: Function succeeded.
 */

bool AwsDoc::DynamoDB::getItem(const Aws::String &tableName,
                               const Aws::String &partitionKey,
                               const Aws::String &partitionValue,
```

```
                                const Aws::Client::ClientConfiguration
 &clientConfiguration) {
    Aws::DynamoDB::DynamoDBClient dynamoClient(clientConfiguration);
    Aws::DynamoDB::Model::GetItemRequest request;

    // Set up the request.
    request.SetTableName(tableName);
    request.AddKey(partitionKey,
                   Aws::DynamoDB::Model::AttributeValue().SetS(partitionValue));

    // Retrieve the item's fields and values.
    const Aws::DynamoDB::Model::GetItemOutcome &outcome =
 dynamoClient.GetItem(request);
    if (outcome.IsSuccess()) {
        // Reference the retrieved fields/values.
        const Aws::Map<Aws::String, Aws::DynamoDB::Model::AttributeValue> &item =
 outcome.GetResult().GetItem();
        if (!item.empty()) {
            // Output each retrieved field and its value.
            for (const auto &i: item)
                std::cout << "Values: " << i.first << ": " << i.second.GetS()
                          << std::endl;
        }
        else {
            std::cout << "No item found with the key " << partitionKey <<
 std::endl;
        }
    }
    else {
        std::cerr << "Failed to get item: " << outcome.GetError().GetMessage();
    }

    return outcome.IsSuccess();
}
```

- For API details, see GetItem in *AWS SDK for C++ API Reference*.

CLI

AWS CLI

Example 1: To read an item in a table

The following `get-item` example retrieves an item from the `MusicCollection` table. The table has a hash-and-range primary key (`Artist` and `SongTitle`), so you must specify both of these attributes. The command also requests information about the read capacity consumed by the operation.

```
aws dynamodb get-item \
    --table-name MusicCollection \
    --key file://key.json \
    --return-consumed-capacity TOTAL
```

Contents of `key.json`:

```
{
    "Artist": {"S": "Acme Band"},
    "SongTitle": {"S": "Happy Day"}
}
```

Output:

```
{
    "Item": {
        "AlbumTitle": {
            "S": "Songs About Life"
        },
        "SongTitle": {
            "S": "Happy Day"
        },
        "Artist": {
            "S": "Acme Band"
        }
    },
    "ConsumedCapacity": {
        "TableName": "MusicCollection",
        "CapacityUnits": 0.5
    }
}
```

For more information, see Reading an Item in the *Amazon DynamoDB Developer Guide*.

Example 2: To read an item using a consistent read

The following example retrieves an item from the `MusicCollection` table using strongly consistent reads.

```
aws dynamodb get-item \
    --table-name MusicCollection \
    --key file://key.json \
    --consistent-read \
    --return-consumed-capacity TOTAL
```

Contents of `key.json`:

```
{
    "Artist": {"S": "Acme Band"},
    "SongTitle": {"S": "Happy Day"}
}
```

Output:

```
{
    "Item": {
        "AlbumTitle": {
            "S": "Songs About Life"
        },
        "SongTitle": {
            "S": "Happy Day"
        },
        "Artist": {
            "S": "Acme Band"
        }
    },
    "ConsumedCapacity": {
        "TableName": "MusicCollection",
        "CapacityUnits": 1.0
    }
}
```

For more information, see Reading an Item in the *Amazon DynamoDB Developer Guide*.

Example 3: To retrieve specific attributes of an item

The following example uses a projection expression to retrieve only three attributes of the desired item.

```
aws dynamodb get-item \
    --table-name ProductCatalog \
    --key '{"Id": {"N": "102"}}' \
    --projection-expression "#T, #C, #P" \
    --expression-attribute-names file://names.json
```

Contents of `names.json`:

```
{
    "#T": "Title",
    "#C": "ProductCategory",
    "#P": "Price"
}
```

Output:

```
{
    "Item": {
        "Price": {
            "N": "20"
        },
        "Title": {
            "S": "Book 102 Title"
        },
        "ProductCategory": {
            "S": "Book"
        }
    }
}
```

For more information, see Reading an Item in the *Amazon DynamoDB Developer Guide*.

- For API details, see GetItem in *AWS CLI Command Reference*.

Go

SDK for Go V2

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import (
 "context"
 "errors"
 "log"
 "time"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/attributevalue"
 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/expression"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb/types"
)

// TableBasics encapsulates the Amazon DynamoDB service actions used in the
 examples.
// It contains a DynamoDB service client that is used to act on the specified
 table.
type TableBasics struct {
 DynamoDbClient *dynamodb.Client
 TableName      string
}



// GetMovie gets movie data from the DynamoDB table by using the primary
 composite key
// made of title and year.
func (basics TableBasics) GetMovie(ctx context.Context, title string, year int)
 (Movie, error) {
 movie := Movie{Title: title, Year: year}
 response, err := basics.DynamoDbClient.GetItem(ctx, &dynamodb.GetItemInput{
```

```
  Key: movie.GetKey(), TableName: aws.String(basics.TableName),
 })
 if err != nil {
  log.Printf("Couldn't get info about %v. Here's why: %v\n", title, err)
 } else {
  err = attributevalue.UnmarshalMap(response.Item, &movie)
  if err != nil {
   log.Printf("Couldn't unmarshal response. Here's why: %v\n", err)
  }
 }
 return movie, err
}
```

Define a Movie struct that is used in this example.

```
import (
 "archive/zip"
 "bytes"
 "encoding/json"
 "fmt"
 "io"
 "log"
 "net/http"

 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/attributevalue"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb/types"
)

// Movie encapsulates data about a movie. Title and Year are the composite
 primary key
// of the movie in Amazon DynamoDB. Title is the sort key, Year is the partition
 key,
// and Info is additional data.
type Movie struct {
 Title string                `dynamodbav:"title"`
 Year  int                   `dynamodbav:"year"`
 Info  map[string]interface{} `dynamodbav:"info"`
}

// GetKey returns the composite primary key of the movie in a format that can be
```

```
// sent to DynamoDB.
func (movie Movie) GetKey() map[string]types.AttributeValue {
 title, err := attributevalue.Marshal(movie.Title)
 if err != nil {
  panic(err)
 }
 year, err := attributevalue.Marshal(movie.Year)
 if err != nil {
  panic(err)
 }
 return map[string]types.AttributeValue{"title": title, "year": year}
}

// String returns the title, year, rating, and plot of a movie, formatted for the
 example.
func (movie Movie) String() string {
 return fmt.Sprintf("%v\n\tReleased: %v\n\tRating: %v\n\tPlot: %v\n",
  movie.Title, movie.Year, movie.Info["rating"], movie.Info["plot"])
}
```

- For API details, see GetItem in *AWS SDK for Go API Reference*.

Java

SDK for Java 2.x

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Gets an item from a table by using the DynamoDbClient.

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.GetItemRequest;
import java.util.HashMap;
```

```
import java.util.Map;
import java.util.Set;

/**
 * Before running this Java V2 code example, set up your development
 * environment, including your credentials.
 *
 * For more information, see the following documentation topic:
 *
 * https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-
started.html
 *
 * To get an item from an Amazon DynamoDB table using the AWS SDK for Java V2,
 * its better practice to use the
 * Enhanced Client, see the EnhancedGetItem example.
 */
public class GetItem {
    public static void main(String[] args) {
        final String usage = """

                Usage:
                    <tableName> <key> <keyVal>

                Where:
                    tableName - The Amazon DynamoDB table from which an item is
 retrieved (for example, Music3).\s
                    key - The key used in the Amazon DynamoDB table (for example,
 Artist).\s
                    keyval - The key value that represents the item to get (for
 example, Famous Band).
                """;

        if (args.length != 3) {
            System.out.println(usage);
            System.exit(1);
        }

        String tableName = args[0];
        String key = args[1];
        String keyVal = args[2];
        System.out.format("Retrieving item \"%s\" from \"%s\"\n", keyVal,
 tableName);
        Region region = Region.US_EAST_1;
        DynamoDbClient ddb = DynamoDbClient.builder()
```

```
                .region(region)
                .build();

        getDynamoDBItem(ddb, tableName, key, keyVal);
        ddb.close();
    }

    public static void getDynamoDBItem(DynamoDbClient ddb, String tableName,
 String key, String keyVal) {
        HashMap<String, AttributeValue> keyToGet = new HashMap<>();
        keyToGet.put(key, AttributeValue.builder()
                .s(keyVal)
                .build());

        GetItemRequest request = GetItemRequest.builder()
                .key(keyToGet)
                .tableName(tableName)
                .build();

        try {
            // If there is no matching item, GetItem does not return any data.
            Map<String, AttributeValue> returnedItem =
 ddb.getItem(request).item();
            if (returnedItem.isEmpty())
                System.out.format("No item found with the key %s!\n", key);
            else {
                Set<String> keys = returnedItem.keySet();
                System.out.println("Amazon DynamoDB table attributes: \n");
                for (String key1 : keys) {
                    System.out.format("%s: %s\n", key1,
 returnedItem.get(key1).toString());
                }
            }

        } catch (DynamoDbException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }
}
```

- For API details, see GetItem in *AWS SDK for Java 2.x API Reference*.

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

This example uses the document client to simplify working with items in DynamoDB. For API details see GetCommand.

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
import { DynamoDBDocumentClient, GetCommand } from "@aws-sdk/lib-dynamodb";

const client = new DynamoDBClient({});
const docClient = DynamoDBDocumentClient.from(client);

export const main = async () => {
  const command = new GetCommand({
    TableName: "AngryAnimals",
    Key: {
      CommonName: "Shoebill",
    },
  });

  const response = await docClient.send(command);
  console.log(response);
  return response;
};
```

- For API details, see GetItem in *AWS SDK for JavaScript API Reference*.

SDK for JavaScript (v2)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Get an item from a table.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create the DynamoDB service object
var ddb = new AWS.DynamoDB({ apiVersion: "2012-08-10" });

var params = {
  TableName: "TABLE",
  Key: {
    KEY_NAME: { N: "001" },
  },
  ProjectionExpression: "ATTRIBUTE_NAME",
};

// Call DynamoDB to read the item from the table
ddb.getItem(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data.Item);
  }
});
```

Get an item from a table using the DynamoDB document client.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create DynamoDB document client
var docClient = new AWS.DynamoDB.DocumentClient({ apiVersion: "2012-08-10" });

var params = {
  TableName: "EPISODES_TABLE",
  Key: { KEY_NAME: VALUE },
};
```

```
docClient.get(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data.Item);
  }
});
```

- For more information, see AWS SDK for JavaScript Developer Guide.
- For API details, see GetItem in *AWS SDK for JavaScript API Reference*.

Kotlin

SDK for Kotlin

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
suspend fun getSpecificItem(
    tableNameVal: String,
    keyName: String,
    keyVal: String,
) {
    val keyToGet = mutableMapOf<String, AttributeValue>()
    keyToGet[keyName] = AttributeValue.S(keyVal)

    val request =
        GetItemRequest {
            key = keyToGet
            tableName = tableNameVal
        }

    DynamoDbClient.fromEnvironment { region = "us-east-1" }.use { ddb ->
        val returnedItem = ddb.getItem(request)
        val numbersMap = returnedItem.item
        numbersMap?.forEach { key1 ->
            println(key1.key)
```

```
            println(key1.value)
        }
    }
}
```

- For API details, see GetItem in *AWS SDK for Kotlin API reference*.

PHP

SDK for PHP

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
        $movie = $service->getItemByKey($tableName, $key);
        echo "\nThe movie {$movie['Item']['title']['S']} was released in
{$movie['Item']['year']['N']}.\n";

    public function getItemByKey(string $tableName, array $key)
    {
        return $this->dynamoDbClient->getItem([
            'Key' => $key['Item'],
            'TableName' => $tableName,
        ]);
    }
```

- For API details, see GetItem in *AWS SDK for PHP API Reference*.

PowerShell

Tools for PowerShell V4

Example 1: Returns the DynamoDB item with the partition key SongTitle and the sort key Artist.

```
$key = @{
  SongTitle = 'Somewhere Down The Road'
  Artist = 'No One You Know'
} | ConvertTo-DDBItem

Get-DDBItem -TableName 'Music' -Key $key | ConvertFrom-DDBItem
```

Output:

```
Name                           Value
----                           -----
Genre                          Country
SongTitle                      Somewhere Down The Road
Price                          1.94
Artist                         No One You Know
CriticRating                   9
AlbumTitle                     Somewhat Famous
```

- For API details, see GetItem in *AWS Tools for PowerShell Cmdlet Reference (V4)*.

Tools for PowerShell V5

Example 1: Returns the DynamoDB item with the partition key SongTitle and the sort key Artist.

```
$key = @{
  SongTitle = 'Somewhere Down The Road'
  Artist = 'No One You Know'
} | ConvertTo-DDBItem

Get-DDBItem -TableName 'Music' -Key $key | ConvertFrom-DDBItem
```

Output:

```
Name                           Value
----                           -----
Genre                          Country
SongTitle                      Somewhere Down The Road
Price                          1.94
Artist                         No One You Know
CriticRating                   9
AlbumTitle                     Somewhat Famous
```

- For API details, see GetItem in *AWS Tools for PowerShell Cmdlet Reference (V5)*.

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
class Movies:
    """Encapsulates an Amazon DynamoDB table of movie data.

    Example data structure for a movie record in this table:
        {
            "year": 1999,
            "title": "For Love of the Game",
            "info": {
                "directors": ["Sam Raimi"],
                "release_date": "1999-09-15T00:00:00Z",
                "rating": 6.3,
                "plot": "A washed up pitcher flashes through his career.",
                "rank": 4987,
                "running_time_secs": 8220,
                "actors": [
                    "Kevin Costner",
                    "Kelly Preston",
                    "John C. Reilly"
                ]
            }
        }
    """

    def __init__(self, dyn_resource):
        """
        :param dyn_resource: A Boto3 DynamoDB resource.
        """
        self.dyn_resource = dyn_resource
        # The table variable is set during the scenario in the call to
        # 'exists' if the table exists. Otherwise, it is set by 'create_table'.
```

```
        self.table = None


    def get_movie(self, title, year):
        """
        Gets movie data from the table for a specific movie.

        :param title: The title of the movie.
        :param year: The release year of the movie.
        :return: The data about the requested movie.
        """
        try:
            response = self.table.get_item(Key={"year": year, "title": title})
        except ClientError as err:
            logger.error(
                "Couldn't get movie %s from table %s. Here's why: %s: %s",
                title,
                self.table.name,
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
        else:
            return response["Item"]
```

- For API details, see GetItem in *AWS SDK for Python (Boto3) API Reference*.

Ruby

SDK for Ruby

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
class DynamoDBBasics
  attr_reader :dynamo_resource, :table
```

```
def initialize(table_name)
  client = Aws::DynamoDB::Client.new(region: 'us-east-1')
  @dynamo_resource = Aws::DynamoDB::Resource.new(client: client)
  @table = @dynamo_resource.table(table_name)
end

# Gets movie data from the table for a specific movie.
#
# @param title [String] The title of the movie.
# @param year [Integer] The release year of the movie.
# @return [Hash] The data about the requested movie.
def get_item(title, year)
  @table.get_item(key: { 'year' => year, 'title' => title })
rescue Aws::DynamoDB::Errors::ServiceError => e
  puts("Couldn't get movie #{title} (#{year}) from table #{@table.name}:\n")
  puts("\t#{e.code}: #{e.message}")
  raise
end
```

- For API details, see GetItem in *AWS SDK for Ruby API Reference*.

SAP ABAP

SDK for SAP ABAP

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
TRY.
    oo_item = lo_dyn->getitem(
      iv_tablename               = iv_table_name
      it_key                     = it_key ).
    DATA(lt_attr) = oo_item->get_item( ).
    DATA(lo_title) = lt_attr[ key = 'title' ]-value.
    DATA(lo_year) = lt_attr[ key = 'year' ]-value.
    DATA(lo_rating) = lt_attr[ key = 'rating' ]-value.
    MESSAGE 'Movie name is: ' && lo_title->get_s( )
      && 'Movie year is: ' && lo_year->get_n( )
```

```
          && 'Moving rating is: ' && lo_rating->get_n( ) TYPE 'I'.
      CATCH /aws1/cx_dynresourcenotfoundex.
        MESSAGE 'The table or index does not exist' TYPE 'E'.
    ENDTRY.
```

- For API details, see GetItem in *AWS SDK for SAP ABAP API reference*.

Swift

SDK for Swift

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import AWSDynamoDB


    /// Return a `Movie` record describing the specified movie from the Amazon
    /// DynamoDB table.
    ///
    /// - Parameters:
    ///   - title: The movie's title (`String`).
    ///   - year: The movie's release year (`Int`).
    ///
    /// - Throws: `MoviesError.ItemNotFound` if the movie isn't in the table.
    ///
    /// - Returns: A `Movie` record with the movie's details.
    func get(title: String, year: Int) async throws -> Movie {
        do {
            guard let client = self.ddbClient else {
                throw MoviesError.UninitializedClient
            }

            let input = GetItemInput(
                key: [
                    "year": .n(String(year)),
                    "title": .s(title)
                ],
```

```
                tableName: self.tableName
            )
            let output = try await client.getItem(input: input)
            guard let item = output.item else {
                throw MoviesError.ItemNotFound
            }

            let movie = try Movie(withItem: item)
            return movie
        } catch {
            print("ERROR: get:", dump(error))
            throw error
        }
    }
```

- For API details, see GetItem in *AWS SDK for Swift API reference*.

For a complete list of AWS SDK developer guides and code examples, see Using DynamoDB with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use `ListTables` with an AWS SDK or CLI

The following code examples show how to use `ListTables`.

Action examples are code excerpts from larger programs and must be run in context. You can see this action in context in the following code example:

- Set up Attribute-Based Access Control

.NET

SDK for .NET

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    private static async Task ListMyTables()
    {
        Console.WriteLine("\n*** Listing tables ***");

        string lastTableNameEvaluated = null;
        do
        {
            var response = await Client.ListTablesAsync(new ListTablesRequest
            {
                Limit = 2,
                ExclusiveStartTableName = lastTableNameEvaluated
            });

            foreach (var name in response.TableNames)
            {
                Console.WriteLine(name);
            }

            lastTableNameEvaluated = response.LastEvaluatedTableName;
        } while (lastTableNameEvaluated != null);
    }
```

- For API details, see ListTables in *AWS SDK for .NET API Reference*.

Bash

AWS CLI with Bash script

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
##############################################################################
# function dynamodb_list_tables
#
# This function lists all the tables in a DynamoDB.
#
# Returns:
```

```
#       0 - If successful.
#       1 - If it fails.
##############################################################################
function dynamodb_list_tables() {
  response=$(aws dynamodb list-tables \
    --output text \
    --query "TableNames")

  local error_code=${?}

  if [[ $error_code -ne 0 ]]; then
    aws_cli_error_log $error_code
    errecho "ERROR: AWS reports batch-write-item operation failed.$response"
    return 1
  fi

  echo "$response" | tr -s "[:space:]" "\n"

  return 0
}
```

The utility functions used in this example.

```
###############################################################################
# function errecho
#
# This function outputs everything sent to it to STDERR (standard error output).
###############################################################################
function errecho() {
  printf "%s\n" "$*" 1>&2
}

###############################################################################
# function aws_cli_error_log()
#
# This function is used to log the error messages from the AWS CLI.
#
# See https://docs.aws.amazon.com/cli/latest/topic/return-codes.html#cli-aws-
help-return-codes.
#
# The function expects the following argument:
#         $1 - The error code returned by the AWS CLI.
```

```
#
#  Returns:
#          0: - Success.
#
###############################################################################
function aws_cli_error_log() {
  local err_code=$1
  errecho "Error code : $err_code"
  if [ "$err_code" == 1 ]; then
    errecho "  One or more S3 transfers failed."
  elif [ "$err_code" == 2 ]; then
    errecho "  Command line failed to parse."
  elif [ "$err_code" == 130 ]; then
    errecho "  Process received SIGINT."
  elif [ "$err_code" == 252 ]; then
    errecho "  Command syntax invalid."
  elif [ "$err_code" == 253 ]; then
    errecho "  The system environment or configuration was invalid."
  elif [ "$err_code" == 254 ]; then
    errecho "  The service returned an error."
  elif [ "$err_code" == 255 ]; then
    errecho "  255 is a catch-all error."
  fi

  return 0
}
```

- For API details, see ListTables in *AWS CLI Command Reference*.

C++

SDK for C++

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
//! List the Amazon DynamoDB tables for the current AWS account.
/*!
```

```
  \sa listTables()
  \param clientConfiguration: AWS client configuration.
  \return bool: Function succeeded.
 */

bool AwsDoc::DynamoDB::listTables(
        const Aws::Client::ClientConfiguration &clientConfiguration) {
    Aws::DynamoDB::DynamoDBClient dynamoClient(clientConfiguration);

    Aws::DynamoDB::Model::ListTablesRequest listTablesRequest;
    listTablesRequest.SetLimit(50);
    do {
        const Aws::DynamoDB::Model::ListTablesOutcome &outcome =
 dynamoClient.ListTables(
                listTablesRequest);
        if (!outcome.IsSuccess()) {
            std::cout << "Error: " << outcome.GetError().GetMessage() <<
 std::endl;
            return false;
        }

        for (const auto &tableName: outcome.GetResult().GetTableNames())
            std::cout << tableName << std::endl;
        listTablesRequest.SetExclusiveStartTableName(
                outcome.GetResult().GetLastEvaluatedTableName());

    } while (!listTablesRequest.GetExclusiveStartTableName().empty());

    return true;
}
```

- For API details, see ListTables in *AWS SDK for C++ API Reference*.

CLI

AWS CLI

Example 1: To list tables

The following `list-tables` example lists all of the tables associated with the current AWS account and Region.

```
aws dynamodb list-tables
```

Output:

```
{
    "TableNames": [
        "Forum",
        "ProductCatalog",
        "Reply",
        "Thread"
    ]
}
```

For more information, see Listing Table Names in the *Amazon DynamoDB Developer Guide*.

Example 2: To limit page size

The following example returns a list of all existing tables, but retrieves only one item in each call, performing multiple calls if necessary to get the entire list. Limiting the page size is useful when running list commands on a large number of resources, which can result in a "timed out" error when using the default page size of 1000.

```
aws dynamodb list-tables \
    --page-size 1
```

Output:

```
{
    "TableNames": [
        "Forum",
        "ProductCatalog",
        "Reply",
        "Thread"
    ]
}
```

For more information, see Listing Table Names in the *Amazon DynamoDB Developer Guide*.

Example 3: To limit the number of items returned

The following example limits the number of items returned to 2. The response includes a `NextToken` value with which to retrieve the next page of results.

```
aws dynamodb list-tables \
    --max-items 2
```

Output:

```
{
    "TableNames": [
        "Forum",
        "ProductCatalog"
    ],
    "NextToken":
 "abCDeFGhiJKlmnOPqrSTuvwxYZ1aBCdEFghijK7LM51nOpqRSTuv3WxY3ZabC5dEFGhI2Jk3LmnoPQ6RST9"
}
```

For more information, see Listing Table Names in the *Amazon DynamoDB Developer Guide*.

Example 4: To retrieve the next page of results

The following command uses the `NextToken` value from a previous call to the `list-tables` command to retrieve another page of results. Since the response in this case does not include a `NextToken` value, we know that we have reached the end of the results.

```
aws dynamodb list-tables \
    --starting-
token abCDeFGhiJKlmnOPqrSTuvwxYZ1aBCdEFghijK7LM51nOpqRSTuv3WxY3ZabC5dEFGhI2Jk3LmnoPQ6RST9
```

Output:

```
{
    "TableNames": [
        "Reply",
        "Thread"
    ]
}
```

For more information, see Listing Table Names in the *Amazon DynamoDB Developer Guide*.

- For API details, see ListTables in *AWS CLI Command Reference*.

Go

SDK for Go V2

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import (
 "context"
 "errors"
 "log"
 "time"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/attributevalue"
 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/expression"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb/types"
)

// TableBasics encapsulates the Amazon DynamoDB service actions used in the
 examples.
// It contains a DynamoDB service client that is used to act on the specified
 table.
type TableBasics struct {
 DynamoDbClient *dynamodb.Client
 TableName      string
}



// ListTables lists the DynamoDB table names for the current account.
func (basics TableBasics) ListTables(ctx context.Context) ([]string, error) {
 var tableNames []string
 var output *dynamodb.ListTablesOutput
 var err error
```

```
 tablePaginator := dynamodb.NewListTablesPaginator(basics.DynamoDbClient,
 &dynamodb.ListTablesInput{})
 for tablePaginator.HasMorePages() {
  output, err = tablePaginator.NextPage(ctx)
  if err != nil {
   log.Printf("Couldn't list tables. Here's why: %v\n", err)
   break
  } else {
   tableNames = append(tableNames, output.TableNames...)
  }
 }
 return tableNames, err
}
```

- For API details, see ListTables in *AWS SDK for Go API Reference*.

Java

SDK for Java 2.x

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.ListTablesRequest;
import software.amazon.awssdk.services.dynamodb.model.ListTablesResponse;
import java.util.List;

/**
 * Before running this Java V2 code example, set up your development
 * environment, including your credentials.
 *
 * For more information, see the following documentation topic:
 *
```

```
 * https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-
started.html
 */
public class ListTables {
    public static void main(String[] args) {
        System.out.println("Listing your Amazon DynamoDB tables:\n");
        Region region = Region.US_EAST_1;
        DynamoDbClient ddb = DynamoDbClient.builder()
                .region(region)
                .build();
        listAllTables(ddb);
        ddb.close();
    }

    public static void listAllTables(DynamoDbClient ddb) {
        boolean moreTables = true;
        String lastName = null;

        while (moreTables) {
            try {
                ListTablesResponse response = null;
                if (lastName == null) {
                    ListTablesRequest request =
 ListTablesRequest.builder().build();
                    response = ddb.listTables(request);
                } else {
                    ListTablesRequest request = ListTablesRequest.builder()
                            .exclusiveStartTableName(lastName).build();
                    response = ddb.listTables(request);
                }

                List<String> tableNames = response.tableNames();
                if (tableNames.size() > 0) {
                    for (String curName : tableNames) {
                        System.out.format("* %s\n", curName);
                    }
                } else {
                    System.out.println("No tables found!");
                    System.exit(0);
                }

                lastName = response.lastEvaluatedTableName();
                if (lastName == null) {
                    moreTables = false;
```

```
                }

            } catch (DynamoDbException e) {
                System.err.println(e.getMessage());
                System.exit(1);
            }
        }
        System.out.println("\nDone!");
    }
}
```

- For API details, see ListTables in *AWS SDK for Java 2.x API Reference*.

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import { ListTablesCommand, DynamoDBClient } from "@aws-sdk/client-dynamodb";

const client = new DynamoDBClient({});

export const main = async () => {
  const command = new ListTablesCommand({});

  const response = await client.send(command);
  console.log(response);
  return response;
};
```

- For more information, see AWS SDK for JavaScript Developer Guide.
- For API details, see ListTables in *AWS SDK for JavaScript API Reference*.

SDK for JavaScript (v2)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create the DynamoDB service object
var ddb = new AWS.DynamoDB({ apiVersion: "2012-08-10" });

// Call DynamoDB to retrieve the list of tables
ddb.listTables({ Limit: 10 }, function (err, data) {
  if (err) {
    console.log("Error", err.code);
  } else {
    console.log("Table names are ", data.TableNames);
  }
});
```

- For more information, see AWS SDK for JavaScript Developer Guide.
- For API details, see ListTables in *AWS SDK for JavaScript API Reference*.

Kotlin

SDK for Kotlin

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
suspend fun listAllTables() {
```

```
    DynamoDbClient.fromEnvironment { region = "us-east-1" }.use { ddb ->
        val response = ddb.listTables(ListTablesRequest {})
        response.tableNames?.forEach { tableName ->
            println("Table name is $tableName")
        }
    }
}
```

- For API details, see ListTables in *AWS SDK for Kotlin API reference*.

PHP

SDK for PHP

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    public function listTables($exclusiveStartTableName = "", $limit = 100)
    {
        $this->dynamoDbClient->listTables([
            'ExclusiveStartTableName' => $exclusiveStartTableName,
            'Limit' => $limit,
        ]);
    }
```

- For API details, see ListTables in *AWS SDK for PHP API Reference*.

PowerShell

Tools for PowerShell V4

Example 1: Returns details of all tables, automatically iterating until the service indicates no further tables exist.

```
Get-DDBTableList
```

- For API details, see ListTables in *AWS Tools for PowerShell Cmdlet Reference (V4)*.

Tools for PowerShell V5

Example 1: Returns details of all tables, automatically iterating until the service indicates no further tables exist.

```
Get-DDBTableList
```

- For API details, see ListTables in *AWS Tools for PowerShell Cmdlet Reference (V5)*.

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
class Movies:
    """Encapsulates an Amazon DynamoDB table of movie data.

    Example data structure for a movie record in this table:
        {
            "year": 1999,
            "title": "For Love of the Game",
            "info": {
                "directors": ["Sam Raimi"],
                "release_date": "1999-09-15T00:00:00Z",
                "rating": 6.3,
                "plot": "A washed up pitcher flashes through his career.",
                "rank": 4987,
                "running_time_secs": 8220,
                "actors": [
                    "Kevin Costner",
                    "Kelly Preston",
                    "John C. Reilly"
                ]
            }
```

```
        }
    """

    def __init__(self, dyn_resource):
        """
        :param dyn_resource: A Boto3 DynamoDB resource.
        """
        self.dyn_resource = dyn_resource
        # The table variable is set during the scenario in the call to
        # 'exists' if the table exists. Otherwise, it is set by 'create_table'.
        self.table = None


    def list_tables(self):
        """
        Lists the Amazon DynamoDB tables for the current account.

        :return: The list of tables.
        """
        try:
            tables = []
            for table in self.dyn_resource.tables.all():
                print(table.name)
                tables.append(table)
        except ClientError as err:
            logger.error(
                "Couldn't list tables. Here's why: %s: %s",
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
        else:
            return tables
```

- For API details, see ListTables in *AWS SDK for Python (Boto3) API Reference*.

Ruby

SDK for Ruby

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Determine whether a table exists.

```
# Encapsulates an Amazon DynamoDB table of movie data.
class Scaffold
  attr_reader :dynamo_resource, :table_name, :table

  def initialize(table_name)
    client = Aws::DynamoDB::Client.new(region: 'us-east-1')
    @dynamo_resource = Aws::DynamoDB::Resource.new(client: client)
    @table_name = table_name
    @table = nil
    @logger = Logger.new($stdout)
    @logger.level = Logger::DEBUG
  end

  # Determines whether a table exists. As a side effect, stores the table in
  # a member variable.
  #
  # @param table_name [String] The name of the table to check.
  # @return [Boolean] True when the table exists; otherwise, False.
  def exists?(table_name)
    @dynamo_resource.client.describe_table(table_name: table_name)
    @logger.debug("Table #{table_name} exists")
  rescue Aws::DynamoDB::Errors::ResourceNotFoundException
    @logger.debug("Table #{table_name} doesn't exist")
    false
  rescue Aws::DynamoDB::Errors::ServiceError => e
    puts("Couldn't check for existence of #{table_name}:\n")
    puts("\t#{e.code}: #{e.message}")
    raise
  end
```

- For API details, see ListTables in *AWS SDK for Ruby API Reference*.

Rust

SDK for Rust

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
pub async fn list_tables(client: &Client) -> Result<Vec<String>, Error> {
    let paginator = client.list_tables().into_paginator().items().send();
    let table_names = paginator.collect::<Result<Vec<_>, _>>().await?;

    println!("Tables:");

    for name in &table_names {
        println!("  {}", name);
    }

    println!("Found {} tables", table_names.len());
    Ok(table_names)
}
```

Determine whether table exists.

```
pub async fn table_exists(client: &Client, table: &str) -> Result<bool, Error> {
    debug!("Checking for table: {table}");
    let table_list = client.list_tables().send().await;

    match table_list {
        Ok(list) => Ok(list.table_names().contains(&table.into())),
        Err(e) => Err(e.into()),
    }
}
```

- For API details, see ListTables in *AWS SDK for Rust API reference*.

SAP ABAP

SDK for SAP ABAP

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    TRY.
        oo_result = lo_dyn->listtables( ).
        " You can loop over the oo_result to get table properties like this.
        LOOP AT oo_result->get_tablenames( ) INTO DATA(lo_table_name).
          DATA(lv_tablename) = lo_table_name->get_value( ).
        ENDLOOP.
        DATA(lv_tablecount) = lines( oo_result->get_tablenames( ) ).
        MESSAGE 'Found ' && lv_tablecount && ' tables' TYPE 'I'.
      CATCH /aws1/cx_rt_service_generic INTO DATA(lo_exception).
        DATA(lv_error) = |"{ lo_exception->av_err_code }" - { lo_exception-
>av_err_msg }|.
        MESSAGE lv_error TYPE 'E'.
    ENDTRY.
```

- For API details, see ListTables in *AWS SDK for SAP ABAP API reference*.

Swift

SDK for Swift

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import AWSDynamoDB
```

```
    /// Get a list of the DynamoDB tables available in the specified Region.
    ///
    /// - Returns: An array of strings listing all of the tables available
    ///   in the Region specified when the session was created.
    public func getTableList() async throws -> [String] {
        let input = ListTablesInput(
        )
        return try await session.listTables(input: input)
    }
```

- For API details, see ListTables in *AWS SDK for Swift API reference*.

For a complete list of AWS SDK developer guides and code examples, see Using DynamoDB with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use `PutItem` with an AWS SDK or CLI

The following code examples show how to use `PutItem`.

Action examples are code excerpts from larger programs and must be run in context. You can see this action in context in the following code examples:

- Learn the basics
- Accelerate reads with DAX
- Advanced Global Secondary Index scenarios
- Create an item with a TTL
- Create and manage global tables demonstrating MREC
- Create and manage MRSC global tables
- Use conditional operations
- Work with global tables and multi-Region replication eventual consistency (MREC)
- Work with Streams and Time-to-Live

.NET

SDK for .NET (v4)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    /// <summary>
    /// Adds a new item to the table.
    /// </summary>
    /// <param name="newMovie">A Movie object containing informtation for
    /// the movie to add to the table.</param>
    /// <param name="tableName">The name of the table where the item will be
 added.</param>
    /// <returns>A Boolean value that indicates the results of adding the item.</
returns>
    public async Task<bool> PutItemAsync(Movie newMovie, string tableName)
    {
        try
        {
            var item = new Dictionary<string, AttributeValue>
            {
                ["title"] = new AttributeValue { S = newMovie.Title },
                ["year"] = new AttributeValue { N = newMovie.Year.ToString() },
            };

            var request = new PutItemRequest
            {
                TableName = tableName,
                Item = item,
            };

            await _amazonDynamoDB.PutItemAsync(request);
            return true;
        }
        catch (ResourceNotFoundException ex)
        {
            Console.WriteLine($"Table {tableName} was not found. {ex.Message}");
```

```
            return false;
        }
        catch (AmazonDynamoDBException ex)
        {
            Console.WriteLine($"An Amazon DynamoDB error occurred while putting
item. {ex.Message}");
            throw;
        }
        catch (Exception ex)
        {
            Console.WriteLine($"An error occurred while putting item.
{ex.Message}");
            throw;
        }
    }
```

- For API details, see PutItem in *AWS SDK for .NET API Reference*.

Bash

AWS CLI with Bash script

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
###############################################################################
# function dynamodb_put_item
#
# This function puts an item into a DynamoDB table.
#
# Parameters:
#       -n table_name  -- The name of the table.
#       -i item  -- Path to json file containing the item values.
#
#  Returns:
#       0 - If successful.
#       1 - If it fails.
```

```
###############################################################################
function dynamodb_put_item() {
  local table_name item response
  local option OPTARG # Required to use getopts command in a function.

  ######################################
  # Function usage explanation
  ######################################
  function usage() {
    echo "function dynamodb_put_item"
    echo "Put an item into a DynamoDB table."
    echo " -n table_name  -- The name of the table."
    echo " -i item  -- Path to json file containing the item values."
    echo ""
  }

  while getopts "n:i:h" option; do
    case "${option}" in
      n) table_name="${OPTARG}" ;;
      i) item="${OPTARG}" ;;
      h)
        usage
        return 0
        ;;
      \?)
        echo "Invalid parameter"
        usage
        return 1
        ;;
    esac
  done
  export OPTIND=1

  if [[ -z "$table_name" ]]; then
    errecho "ERROR: You must provide a table name with the -n parameter."
    usage
    return 1
  fi

  if [[ -z "$item" ]]; then
    errecho "ERROR: You must provide an item with the -i parameter."
    usage
    return 1
  fi
```

```
  iecho "Parameters:\n"
  iecho "    table_name:   $table_name"
  iecho "    item:   $item"
  iecho ""
  iecho ""

  response=$(aws dynamodb put-item \
    --table-name "$table_name" \
    --item file://"$item")

  local error_code=${?}

  if [[ $error_code -ne 0 ]]; then
    aws_cli_error_log $error_code
    errecho "ERROR: AWS reports put-item operation failed.$response"
    return 1
  fi

  return 0

}
```

The utility functions used in this example.

```
###############################################################################
# function iecho
#
# This function enables the script to display the specified text only if
# the global variable $VERBOSE is set to true.
###############################################################################
function iecho() {
  if [[ $VERBOSE == true ]]; then
    echo "$@"
  fi
}

###############################################################################
# function errecho
#
# This function outputs everything sent to it to STDERR (standard error output).
###############################################################################
```

```
function errecho() {
  printf "%s\n" "$*" 1>&2
}

###############################################################################
# function aws_cli_error_log()
#
# This function is used to log the error messages from the AWS CLI.
#
# See https://docs.aws.amazon.com/cli/latest/topic/return-codes.html#cli-aws-
help-return-codes.
#
# The function expects the following argument:
#         $1 - The error code returned by the AWS CLI.
#
#  Returns:
#          0: - Success.
#
###############################################################################
function aws_cli_error_log() {
  local err_code=$1
  errecho "Error code : $err_code"
  if [ "$err_code" == 1 ]; then
    errecho "  One or more S3 transfers failed."
  elif [ "$err_code" == 2 ]; then
    errecho "  Command line failed to parse."
  elif [ "$err_code" == 130 ]; then
    errecho "  Process received SIGINT."
  elif [ "$err_code" == 252 ]; then
    errecho "  Command syntax invalid."
  elif [ "$err_code" == 253 ]; then
    errecho "  The system environment or configuration was invalid."
  elif [ "$err_code" == 254 ]; then
    errecho "  The service returned an error."
  elif [ "$err_code" == 255 ]; then
    errecho "  255 is a catch-all error."
  fi

  return 0
}
```

- For API details, see PutItem in *AWS CLI Command Reference*.

C++

SDK for C++

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
//! Put an item in an Amazon DynamoDB table.
/*!
  \sa putItem()
  \param tableName: The table name.
  \param artistKey: The artist key. This is the partition key for the table.
  \param artistValue: The artist value.
  \param albumTitleKey: The album title key.
  \param albumTitleValue: The album title value.
  \param awardsKey: The awards key.
  \param awardsValue: The awards value.
  \param songTitleKey: The song title key.
  \param songTitleValue: The song title value.
  \param clientConfiguration: AWS client configuration.
  \return bool: Function succeeded.
 */
bool AwsDoc::DynamoDB::putItem(const Aws::String &tableName,
                               const Aws::String &artistKey,
                               const Aws::String &artistValue,
                               const Aws::String &albumTitleKey,
                               const Aws::String &albumTitleValue,
                               const Aws::String &awardsKey,
                               const Aws::String &awardsValue,
                               const Aws::String &songTitleKey,
                               const Aws::String &songTitleValue,
                               const Aws::Client::ClientConfiguration
 &clientConfiguration) {
    Aws::DynamoDB::DynamoDBClient dynamoClient(clientConfiguration);

    Aws::DynamoDB::Model::PutItemRequest putItemRequest;
    putItemRequest.SetTableName(tableName);
```

```
    putItemRequest.AddItem(artistKey,
 Aws::DynamoDB::Model::AttributeValue().SetS(
            artistValue)); // This is the hash key.
    putItemRequest.AddItem(albumTitleKey,
 Aws::DynamoDB::Model::AttributeValue().SetS(
            albumTitleValue));
    putItemRequest.AddItem(awardsKey,

 Aws::DynamoDB::Model::AttributeValue().SetS(awardsValue));
    putItemRequest.AddItem(songTitleKey,

 Aws::DynamoDB::Model::AttributeValue().SetS(songTitleValue));

    const Aws::DynamoDB::Model::PutItemOutcome outcome = dynamoClient.PutItem(
            putItemRequest);
    if (outcome.IsSuccess()) {
        std::cout << "Successfully added Item!" << std::endl;
    }
    else {
        std::cerr << outcome.GetError().GetMessage() << std::endl;
        return false;
    }

    return waitTableActive(tableName, dynamoClient);
}
```

Code that waits for the table to become active.

```
//! Query a newly created DynamoDB table until it is active.
/*!
  \sa waitTableActive()
  \param waitTableActive: The DynamoDB table's name.
  \param dynamoClient: A DynamoDB client.
  \return bool: Function succeeded.
*/
bool AwsDoc::DynamoDB::waitTableActive(const Aws::String &tableName,
                                       const Aws::DynamoDB::DynamoDBClient
 &dynamoClient) {

    // Repeatedly call DescribeTable until table is ACTIVE.
    const int MAX_QUERIES = 20;
```

```
    Aws::DynamoDB::Model::DescribeTableRequest request;
    request.SetTableName(tableName);

    int count = 0;
    while (count < MAX_QUERIES) {
        const Aws::DynamoDB::Model::DescribeTableOutcome &result =
 dynamoClient.DescribeTable(
                request);
        if (result.IsSuccess()) {
            Aws::DynamoDB::Model::TableStatus status =
 result.GetResult().GetTable().GetTableStatus();

            if (Aws::DynamoDB::Model::TableStatus::ACTIVE != status) {
                std::this_thread::sleep_for(std::chrono::seconds(1));
            }
            else {
                return true;
            }
        }
        else {
            std::cerr << "Error DynamoDB::waitTableActive "
                      << result.GetError().GetMessage() << std::endl;
            return false;
        }
        count++;
    }
    return false;
}
```

- For API details, see PutItem in *AWS SDK for C++ API Reference*.

CLI

AWS CLI

Example 1: To add an item to a table

The following `put-item` example adds a new item to the *MusicCollection* table.

```
aws dynamodb put-item \
    --table-name MusicCollection \
    --item file://item.json \
```

```
    --return-consumed-capacity TOTAL \
    --return-item-collection-metrics SIZE
```

Contents of `item.json`:

```
{
    "Artist": {"S": "No One You Know"},
    "SongTitle": {"S": "Call Me Today"},
    "AlbumTitle": {"S": "Greatest Hits"}
}
```

Output:

```
{
    "ConsumedCapacity": {
        "TableName": "MusicCollection",
        "CapacityUnits": 1.0
    },
    "ItemCollectionMetrics": {
        "ItemCollectionKey": {
            "Artist": {
                "S": "No One You Know"
            }
        },
        "SizeEstimateRangeGB": [
            0.0,
            1.0
        ]
    }
}
```

For more information, see Writing an Item in the *Amazon DynamoDB Developer Guide*.

Example 2: To conditionally overwrite an item in a table

The following `put-item` example overwrites an existing item in the `MusicCollection` table only if that existing item has an `AlbumTitle` attribute with a value of `Greatest Hits`. The command returns the previous value of the item.

```
aws dynamodb put-item \
    --table-name MusicCollection \
    --item file://item.json \
```

```
    --condition-expression "#A = :A" \
    --expression-attribute-names file://names.json \
    --expression-attribute-values file://values.json \
    --return-values ALL_OLD
```

Contents of `item.json`:

```
{
    "Artist": {"S": "No One You Know"},
    "SongTitle": {"S": "Call Me Today"},
    "AlbumTitle": {"S": "Somewhat Famous"}
}
```

Contents of `names.json`:

```
{
    "#A": "AlbumTitle"
}
```

Contents of `values.json`:

```
{
    ":A": {"S": "Greatest Hits"}
}
```

Output:

```
{
    "Attributes": {
        "AlbumTitle": {
            "S": "Greatest Hits"
        },
        "Artist": {
            "S": "No One You Know"
        },
        "SongTitle": {
            "S": "Call Me Today"
        }
    }
}
```

If the key already exists, you should see the following output:

```
A client error (ConditionalCheckFailedException) occurred when calling the
 PutItem operation: The conditional request failed.
```

For more information, see Writing an Item in the *Amazon DynamoDB Developer Guide*.

- For API details, see PutItem in *AWS CLI Command Reference*.

Go

SDK for Go V2

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import (
 "context"
 "errors"
 "log"
 "time"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/attributevalue"
 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/expression"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb/types"
)

// TableBasics encapsulates the Amazon DynamoDB service actions used in the
 examples.
// It contains a DynamoDB service client that is used to act on the specified
 table.
type TableBasics struct {
 DynamoDbClient *dynamodb.Client
 TableName      string
}
```

```
// AddMovie adds a movie the DynamoDB table.
func (basics TableBasics) AddMovie(ctx context.Context, movie Movie) error {
 item, err := attributevalue.MarshalMap(movie)
 if err != nil {
  panic(err)
 }
 _, err = basics.DynamoDbClient.PutItem(ctx, &dynamodb.PutItemInput{
  TableName: aws.String(basics.TableName), Item: item,
 })
 if err != nil {
  log.Printf("Couldn't add item to table. Here's why: %v\n", err)
 }
 return err
}
```

Define a Movie struct that is used in this example.

```
import (
 "archive/zip"
 "bytes"
 "encoding/json"
 "fmt"
 "io"
 "log"
 "net/http"

 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/attributevalue"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb/types"
)

// Movie encapsulates data about a movie. Title and Year are the composite
 primary key
// of the movie in Amazon DynamoDB. Title is the sort key, Year is the partition
 key,
// and Info is additional data.
type Movie struct {
 Title string                 `dynamodbav:"title"`
 Year  int                    `dynamodbav:"year"`
 Info  map[string]interface{} `dynamodbav:"info"`
```

```
}

// GetKey returns the composite primary key of the movie in a format that can be
// sent to DynamoDB.
func (movie Movie) GetKey() map[string]types.AttributeValue {
 title, err := attributevalue.Marshal(movie.Title)
 if err != nil {
  panic(err)
 }
 year, err := attributevalue.Marshal(movie.Year)
 if err != nil {
  panic(err)
 }
 return map[string]types.AttributeValue{"title": title, "year": year}
}

// String returns the title, year, rating, and plot of a movie, formatted for the
 example.
func (movie Movie) String() string {
 return fmt.Sprintf("%v\n\tReleased: %v\n\tRating: %v\n\tPlot: %v\n",
  movie.Title, movie.Year, movie.Info["rating"], movie.Info["plot"])
}
```

- For API details, see PutItem in *AWS SDK for Go API Reference*.

Java

SDK for Java 2.x

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Puts an item into a table using DynamoDbClient.

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
```

```
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.PutItemRequest;
import software.amazon.awssdk.services.dynamodb.model.PutItemResponse;
import software.amazon.awssdk.services.dynamodb.model.ResourceNotFoundException;
import java.util.HashMap;

/**
 * Before running this Java V2 code example, set up your development
 * environment, including your credentials.
 *
 * For more information, see the following documentation topic:
 *
 * https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-
started.html
 *
 * To place items into an Amazon DynamoDB table using the AWS SDK for Java V2,
 * its better practice to use the
 * Enhanced Client. See the EnhancedPutItem example.
 */
public class PutItem {
    public static void main(String[] args) {
        final String usage = """

                Usage:
                    <tableName> <key> <keyVal> <albumtitle> <albumtitleval>
 <awards> <awardsval> <Songtitle> <songtitleval>

                Where:
                    tableName - The Amazon DynamoDB table in which an item is
 placed (for example, Music3).
                    key - The key used in the Amazon DynamoDB table (for example,
 Artist).
                    keyval - The key value that represents the item to get (for
 example, Famous Band).
                    albumTitle - The Album title (for example, AlbumTitle).
                    AlbumTitleValue - The name of the album (for example, Songs
 About Life ).
                    Awards - The awards column (for example, Awards).
                    AwardVal - The value of the awards (for example, 10).
                    SongTitle - The song title (for example, SongTitle).
                    SongTitleVal - The value of the song title (for example,
 Happy Day).
                **Warning** This program will  place an item that you specify
 into a table!
```

```
                """;

        if (args.length != 9) {
            System.out.println(usage);
            System.exit(1);
        }

        String tableName = args[0];
        String key = args[1];
        String keyVal = args[2];
        String albumTitle = args[3];
        String albumTitleValue = args[4];
        String awards = args[5];
        String awardVal = args[6];
        String songTitle = args[7];
        String songTitleVal = args[8];

        Region region = Region.US_EAST_1;
        DynamoDbClient ddb = DynamoDbClient.builder()
                .region(region)
                .build();

        putItemInTable(ddb, tableName, key, keyVal, albumTitle, albumTitleValue,
awards, awardVal, songTitle,
                songTitleVal);
        System.out.println("Done!");
        ddb.close();
    }

    public static void putItemInTable(DynamoDbClient ddb,
            String tableName,
            String key,
            String keyVal,
            String albumTitle,
            String albumTitleValue,
            String awards,
            String awardVal,
            String songTitle,
            String songTitleVal) {

        HashMap<String, AttributeValue> itemValues = new HashMap<>();
        itemValues.put(key, AttributeValue.builder().s(keyVal).build());
        itemValues.put(songTitle,
AttributeValue.builder().s(songTitleVal).build());
```

```
        itemValues.put(albumTitle,
 AttributeValue.builder().s(albumTitleValue).build());
        itemValues.put(awards, AttributeValue.builder().s(awardVal).build());

        PutItemRequest request = PutItemRequest.builder()
                .tableName(tableName)
                .item(itemValues)
                .build();

        try {
            PutItemResponse response = ddb.putItem(request);
            System.out.println(tableName + " was successfully updated. The
 request id is "
                    + response.responseMetadata().requestId());

        } catch (ResourceNotFoundException e) {
            System.err.format("Error: The Amazon DynamoDB table \"%s\" can't be
 found.\n", tableName);
            System.err.println("Be sure that it exists and that you've typed its
 name correctly!");
            System.exit(1);
        } catch (DynamoDbException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }
}
```

- For API details, see PutItem in *AWS SDK for Java 2.x API Reference*.

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

This example uses the document client to simplify working with items in DynamoDB. For API details see PutCommand.

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
import { PutCommand, DynamoDBDocumentClient } from "@aws-sdk/lib-dynamodb";

const client = new DynamoDBClient({});
const docClient = DynamoDBDocumentClient.from(client);

export const main = async () => {
  const command = new PutCommand({
    TableName: "HappyAnimals",
    Item: {
      CommonName: "Shiba Inu",
    },
  });

  const response = await docClient.send(command);
  console.log(response);
  return response;
};
```

- For API details, see PutItem in *AWS SDK for JavaScript API Reference*.

SDK for JavaScript (v2)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Put an item in a table.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create the DynamoDB service object
var ddb = new AWS.DynamoDB({ apiVersion: "2012-08-10" });
```

```
var params = {
  TableName: "CUSTOMER_LIST",
  Item: {
    CUSTOMER_ID: { N: "001" },
    CUSTOMER_NAME: { S: "Richard Roe" },
  },
};

// Call DynamoDB to add the item to the table
ddb.putItem(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data);
  }
});
```

Put an item in a table using the DynamoDB document client.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create DynamoDB document client
var docClient = new AWS.DynamoDB.DocumentClient({ apiVersion: "2012-08-10" });

var params = {
  TableName: "TABLE",
  Item: {
    HASHKEY: VALUE,
    ATTRIBUTE_1: "STRING_VALUE",
    ATTRIBUTE_2: VALUE_2,
  },
};

docClient.put(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data);
```

```
  }
});
```

- For more information, see AWS SDK for JavaScript Developer Guide.
- For API details, see PutItem in *AWS SDK for JavaScript API Reference*.

Kotlin

SDK for Kotlin

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
suspend fun putItemInTable(
    tableNameVal: String,
    key: String,
    keyVal: String,
    albumTitle: String,
    albumTitleValue: String,
    awards: String,
    awardVal: String,
    songTitle: String,
    songTitleVal: String,
) {
    val itemValues = mutableMapOf<String, AttributeValue>()

    // Add all content to the table.
    itemValues[key] = AttributeValue.S(keyVal)
    itemValues[songTitle] = AttributeValue.S(songTitleVal)
    itemValues[albumTitle] = AttributeValue.S(albumTitleValue)
    itemValues[awards] = AttributeValue.S(awardVal)

    val request =
        PutItemRequest {
            tableName = tableNameVal
            item = itemValues
        }
```

```
    DynamoDbClient.fromEnvironment { region = "us-east-1" }.use { ddb ->
        ddb.putItem(request)
        println(" A new item was placed into $tableNameVal.")
    }
}
```

- For API details, see PutItem in *AWS SDK for Kotlin API reference*.

PHP

SDK for PHP

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
        echo "What's the name of the last movie you watched?\n";
        while (empty($movieName)) {
            $movieName = testable_readline("Movie name: ");
        }
        echo "And what year was it released?\n";
        $movieYear = "year";
        while (!is_numeric($movieYear) || intval($movieYear) != $movieYear) {
            $movieYear = testable_readline("Year released: ");
        }

        $service->putItem([
            'Item' => [
                'year' => [
                    'N' => "$movieYear",
                ],
                'title' => [
                    'S' => $movieName,
                ],
            ],
            'TableName' => $tableName,
        ]);
```

```
    public function putItem(array $array)
    {
        $this->dynamoDbClient->putItem($array);
    }
```

- For API details, see PutItem in *AWS SDK for PHP API Reference*.

PowerShell

Tools for PowerShell V4

Example 1: Creates a new item, or replaces an existing item with a new item.

```
$item = @{
  SongTitle = 'Somewhere Down The Road'
  Artist = 'No One You Know'
        AlbumTitle = 'Somewhat Famous'
        Price = 1.94
        Genre = 'Country'
        CriticRating = 9.0
} | ConvertTo-DDBItem
Set-DDBItem -TableName 'Music' -Item $item
```

- For API details, see PutItem in *AWS Tools for PowerShell Cmdlet Reference (V4)*.

Tools for PowerShell V5

Example 1: Creates a new item, or replaces an existing item with a new item.

```
$item = @{
  SongTitle = 'Somewhere Down The Road'
  Artist = 'No One You Know'
        AlbumTitle = 'Somewhat Famous'
        Price = 1.94
        Genre = 'Country'
        CriticRating = 9.0
} | ConvertTo-DDBItem
Set-DDBItem -TableName 'Music' -Item $item
```

- For API details, see PutItem in *AWS Tools for PowerShell Cmdlet Reference (V5)*.

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
class Movies:
    """Encapsulates an Amazon DynamoDB table of movie data.

    Example data structure for a movie record in this table:
        {
            "year": 1999,
            "title": "For Love of the Game",
            "info": {
                "directors": ["Sam Raimi"],
                "release_date": "1999-09-15T00:00:00Z",
                "rating": 6.3,
                "plot": "A washed up pitcher flashes through his career.",
                "rank": 4987,
                "running_time_secs": 8220,
                "actors": [
                    "Kevin Costner",
                    "Kelly Preston",
                    "John C. Reilly"
                ]
            }
        }
    """

    def __init__(self, dyn_resource):
        """
        :param dyn_resource: A Boto3 DynamoDB resource.
        """
        self.dyn_resource = dyn_resource
        # The table variable is set during the scenario in the call to
        # 'exists' if the table exists. Otherwise, it is set by 'create_table'.
        self.table = None
```

```
    def add_movie(self, title, year, plot, rating):
        """
        Adds a movie to the table.

        :param title: The title of the movie.
        :param year: The release year of the movie.
        :param plot: The plot summary of the movie.
        :param rating: The quality rating of the movie.
        """
        try:
            self.table.put_item(
                Item={
                    "year": year,
                    "title": title,
                    "info": {"plot": plot, "rating": Decimal(str(rating))},
                }
            )
        except ClientError as err:
            logger.error(
                "Couldn't add movie %s to table %s. Here's why: %s: %s",
                title,
                self.table.name,
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
```

- For API details, see PutItem in *AWS SDK for Python (Boto3) API Reference*.

Ruby

SDK for Ruby

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
class DynamoDBBasics
  attr_reader :dynamo_resource, :table

  def initialize(table_name)
    client = Aws::DynamoDB::Client.new(region: 'us-east-1')
    @dynamo_resource = Aws::DynamoDB::Resource.new(client: client)
    @table = @dynamo_resource.table(table_name)
  end

  # Adds a movie to the table.
  #
  # @param movie [Hash] The title, year, plot, and rating of the movie.
  def add_item(movie)
    @table.put_item(
      item: {
        'year' => movie[:year],
        'title' => movie[:title],
        'info' => { 'plot' => movie[:plot], 'rating' => movie[:rating] }
      }
    )
  rescue Aws::DynamoDB::Errors::ServiceError => e
    puts("Couldn't add movie #{title} to table #{@table.name}. Here's why:")
    puts("\t#{e.code}: #{e.message}")
    raise
  end
```

- For API details, see PutItem in *AWS SDK for Ruby API Reference*.

Rust

SDK for Rust

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
pub async fn add_item(client: &Client, item: Item, table: &String) ->
 Result<ItemOut, Error> {
```

```
    let user_av = AttributeValue::S(item.username);
    let type_av = AttributeValue::S(item.p_type);
    let age_av = AttributeValue::S(item.age);
    let first_av = AttributeValue::S(item.first);
    let last_av = AttributeValue::S(item.last);

    let request = client
        .put_item()
        .table_name(table)
        .item("username", user_av)
        .item("account_type", type_av)
        .item("age", age_av)
        .item("first_name", first_av)
        .item("last_name", last_av);

    println!("Executing request [{request:?}] to add item...");

    let resp = request.send().await?;

    let attributes = resp.attributes().unwrap();

    let username = attributes.get("username").cloned();
    let first_name = attributes.get("first_name").cloned();
    let last_name = attributes.get("last_name").cloned();
    let age = attributes.get("age").cloned();
    let p_type = attributes.get("p_type").cloned();

    println!(
        "Added user {:?}, {:?} {:?}, age {:?} as {:?} user",
        username, first_name, last_name, age, p_type
    );

    Ok(ItemOut {
        p_type,
        age,
        username,
        first_name,
        last_name,
    })
}
```

- For API details, see PutItem in *AWS SDK for Rust API reference*.

SAP ABAP

SDK for SAP ABAP

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    TRY.
        DATA(lo_resp) = lo_dyn->putitem(
          iv_tablename = iv_table_name
          it_item      = it_item ).
        MESSAGE '1 row inserted into DynamoDB Table' && iv_table_name TYPE 'I'.
      CATCH /aws1/cx_dyncondalcheckfaile00.
        MESSAGE 'A condition specified in the operation could not be evaluated.'
TYPE 'E'.
      CATCH /aws1/cx_dynresourcenotfoundex.
        MESSAGE 'The table or index does not exist' TYPE 'E'.
      CATCH /aws1/cx_dyntransactconflictex.
        MESSAGE 'Another transaction is using the item' TYPE 'E'.
    ENDTRY.
```

- For API details, see PutItem in *AWS SDK for SAP ABAP API reference*.

Swift

SDK for Swift

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import AWSDynamoDB
```

```
    /// Add a movie specified as a `Movie` structure to the Amazon DynamoDB
    /// table.
    ///
    /// - Parameter movie: The `Movie` to add to the table.
    ///
    func add(movie: Movie) async throws {
        do {
            guard let client = self.ddbClient else {
                throw MoviesError.UninitializedClient
            }

            // Get a DynamoDB item containing the movie data.
            let item = try await movie.getAsItem()

            // Send the `PutItem` request to Amazon DynamoDB.

            let input = PutItemInput(
                item: item,
                tableName: self.tableName
            )
            _ = try await client.putItem(input: input)
        } catch {
            print("ERROR: add movie:", dump(error))
            throw error
        }
    }


    ///
    /// Return an array mapping attribute names to Amazon DynamoDB attribute
    /// values, representing the contents of the `Movie` record as a DynamoDB
    /// item.
    ///
    /// - Returns: The movie item as an array of type
    ///   `[Swift.String:DynamoDBClientTypes.AttributeValue]`.
    ///
    func getAsItem() async throws ->
[Swift.String:DynamoDBClientTypes.AttributeValue]  {
        // Build the item record, starting with the year and title, which are
        // always present.

        var item: [Swift.String:DynamoDBClientTypes.AttributeValue] = [
            "year": .n(String(self.year)),
            "title": .s(self.title)
```

```
        ]

        // Add the `info` field with the rating and/or plot if they're
        // available.

        var details: [Swift.String:DynamoDBClientTypes.AttributeValue] = [:]
        if (self.info.rating != nil || self.info.plot != nil) {
            if self.info.rating != nil {
                details["rating"] = .n(String(self.info.rating!))
            }
            if self.info.plot != nil {
                details["plot"] = .s(self.info.plot!)
            }
        }
        item["info"] = .m(details)

        return item
    }
```

- For API details, see PutItem in *AWS SDK for Swift API reference*.

For a complete list of AWS SDK developer guides and code examples, see Using DynamoDB with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use `Query` with an AWS SDK or CLI

The following code examples show how to use `Query`.

Action examples are code excerpts from larger programs and must be run in context. You can see this action in context in the following code examples:

- Learn the basics
- Accelerate reads with DAX
- Advanced Global Secondary Index scenarios
- Compare multiple values with a single attribute
- Manage Global Secondary Indexes
- Perform advanced query operations
- Query a table using a begins_with condition

- Query a table using a date range
- Query a table using a Global Secondary Index
- Query a table with a complex filter expression
- Query a table with a dynamic filter expression
- Query a table with a filter expression and limit
- Query a table with nested attributes
- Query a table with pagination
- Query a table with strongly consistent reads
- Query for TTL items
- Query tables using date and time patterns
- Use expression attribute names
- Work with Local Secondary Indexes

.NET

SDK for .NET (v4)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
/// <summary>
/// Queries the table for movies released in a particular year and
/// then displays the information for the movies returned.
/// </summary>
/// <param name="tableName">The name of the table to query.</param>
/// <param name="year">The release year for which we want to
/// view movies.</param>
/// <returns>The number of movies that match the query.</returns>
public async Task<int> QueryMoviesAsync(string tableName, int year)
{
    try
    {
        var movieTable = new TableBuilder(_amazonDynamoDB, tableName)
```

```
                .AddHashKey("year", DynamoDBEntryType.Numeric)
                .AddRangeKey("title", DynamoDBEntryType.String)
                .Build();

            var filter = new QueryFilter("year", QueryOperator.Equal, year);

            Console.WriteLine("\nFind movies released in: {year}:");

            var config = new QueryOperationConfig()
            {
                Limit = 10, // 10 items per page.
                Select = SelectValues.SpecificAttributes,
                AttributesToGet = new List<string>
                {
                    "title",
                    "year",
                },
                ConsistentRead = true,
                Filter = filter,
            };

            // Value used to track how many movies match the
            // supplied criteria.
            var moviesFound = 0;

            var search = movieTable.Query(config);
            do
            {
                var movieList = await search.GetNextSetAsync();
                moviesFound += movieList.Count;

                foreach (var movie in movieList)
                {
                    DisplayDocument(movie);
                }
            }
            while (!search.IsDone);

            return moviesFound;
        }
        catch (ResourceNotFoundException ex)
        {
            Console.WriteLine($"Table {tableName} was not found. {ex.Message}");
            return 0;
```

```
        }
        catch (AmazonDynamoDBException ex)
        {
            Console.WriteLine($"An Amazon DynamoDB error occurred while querying
movies. {ex.Message}");
            throw;
        }
        catch (Exception ex)
        {
            Console.WriteLine($"An error occurred while querying movies.
{ex.Message}");
            throw;
        }
    }
```

- For API details, see Query in *AWS SDK for .NET API Reference*.

Bash

AWS CLI with Bash script

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
###############################################################################
# function dynamodb_query
#
# This function queries a DynamoDB table.
#
# Parameters:
#       -n table_name  -- The name of the table.
#       -k key_condition_expression -- The key condition expression.
#       -a attribute_names -- Path to JSON file containing the attribute names.
#       -v attribute_values -- Path to JSON file containing the attribute values.
#       [-p projection_expression]  -- Optional projection expression.
#
#  Returns:
```

```
#       The items as json output.
#  And:
#       0 - If successful.
#       1 - If it fails.
###############################################################################
function dynamodb_query() {
  local table_name key_condition_expression attribute_names attribute_values
 projection_expression response
  local option OPTARG # Required to use getopts command in a function.

  # ######################################
  # Function usage explanation
  ######################################
  function usage() {
    echo "function dynamodb_query"
    echo "Query a DynamoDB table."
    echo " -n table_name  -- The name of the table."
    echo " -k key_condition_expression -- The key condition expression."
    echo " -a attribute_names -- Path to JSON file containing the attribute
 names."
    echo " -v attribute_values -- Path to JSON file containing the attribute
 values."
    echo " [-p projection_expression]  -- Optional projection expression."
    echo ""
  }

  while getopts "n:k:a:v:p:h" option; do
    case "${option}" in
      n) table_name="${OPTARG}" ;;
      k) key_condition_expression="${OPTARG}" ;;
      a) attribute_names="${OPTARG}" ;;
      v) attribute_values="${OPTARG}" ;;
      p) projection_expression="${OPTARG}" ;;
      h)
        usage
        return 0
        ;;
      \?)
        echo "Invalid parameter"
        usage
        return 1
        ;;
    esac
  done
```

```
  export OPTIND=1

  if [[ -z "$table_name" ]]; then
    errecho "ERROR: You must provide a table name with the -n parameter."
    usage
    return 1
  fi

  if [[ -z "$key_condition_expression" ]]; then
    errecho "ERROR: You must provide a key condition expression with the -k
parameter."
    usage
    return 1
  fi

  if [[ -z "$attribute_names" ]]; then
    errecho "ERROR: You must provide a attribute names with the -a parameter."
    usage
    return 1
  fi

  if [[ -z "$attribute_values" ]]; then
    errecho "ERROR: You must provide a attribute values with the -v parameter."
    usage
    return 1
  fi

  if [[ -z "$projection_expression" ]]; then
    response=$(aws dynamodb query \
      --table-name "$table_name" \
      --key-condition-expression "$key_condition_expression" \
      --expression-attribute-names file://"$attribute_names" \
      --expression-attribute-values file://"$attribute_values")
  else
    response=$(aws dynamodb query \
      --table-name "$table_name" \
      --key-condition-expression "$key_condition_expression" \
      --expression-attribute-names file://"$attribute_names" \
      --expression-attribute-values file://"$attribute_values" \
      --projection-expression "$projection_expression")
  fi

  local error_code=${?}
```

```
  if [[ $error_code -ne 0 ]]; then
    aws_cli_error_log $error_code
    errecho "ERROR: AWS reports query operation failed.$response"
    return 1
  fi

  echo "$response"

  return 0
}
```

The utility functions used in this example.

```
###############################################################################
# function errecho
#
# This function outputs everything sent to it to STDERR (standard error output).
###############################################################################
function errecho() {
  printf "%s\n" "$*" 1>&2
}

###############################################################################
# function aws_cli_error_log()
#
# This function is used to log the error messages from the AWS CLI.
#
# See https://docs.aws.amazon.com/cli/latest/topic/return-codes.html#cli-aws-
help-return-codes.
#
# The function expects the following argument:
#         $1 - The error code returned by the AWS CLI.
#
#  Returns:
#          0: - Success.
#
###############################################################################
function aws_cli_error_log() {
  local err_code=$1
  errecho "Error code : $err_code"
  if [ "$err_code" == 1 ]; then
    errecho "  One or more S3 transfers failed."
```

```
  elif [ "$err_code" == 2 ]; then
    errecho "  Command line failed to parse."
  elif [ "$err_code" == 130 ]; then
    errecho "  Process received SIGINT."
  elif [ "$err_code" == 252 ]; then
    errecho "  Command syntax invalid."
  elif [ "$err_code" == 253 ]; then
    errecho "  The system environment or configuration was invalid."
  elif [ "$err_code" == 254 ]; then
    errecho "  The service returned an error."
  elif [ "$err_code" == 255 ]; then
    errecho "  255 is a catch-all error."
  fi

  return 0
}
```

- For API details, see Query in *AWS CLI Command Reference*.

C++

SDK for C++

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
//! Perform a query on an Amazon DynamoDB Table and retrieve items.
/*!
  \sa queryItem()
  \param tableName: The table name.
  \param partitionKey: The partition key.
  \param partitionValue: The value for the partition key.
  \param projectionExpression: The projections expression, which is ignored if
 empty.
  \param clientConfiguration: AWS client configuration.
  \return bool: Function succeeded.
  */
```

```
/*
 * The partition key attribute is searched with the specified value. By default,
 all fields and values
 * contained in the item are returned. If an optional projection expression is
 * specified on the command line, only the specified fields and values are
 * returned.
 */

bool AwsDoc::DynamoDB::queryItems(const Aws::String &tableName,
                                  const Aws::String &partitionKey,
                                  const Aws::String &partitionValue,
                                  const Aws::String &projectionExpression,
                                  const Aws::Client::ClientConfiguration
 &clientConfiguration) {
    Aws::DynamoDB::DynamoDBClient dynamoClient(clientConfiguration);
    Aws::DynamoDB::Model::QueryRequest request;

    request.SetTableName(tableName);

    if (!projectionExpression.empty()) {
        request.SetProjectionExpression(projectionExpression);
    }

    // Set query key condition expression.
    request.SetKeyConditionExpression(partitionKey + "= :valueToMatch");

    // Set Expression AttributeValues.
    Aws::Map<Aws::String, Aws::DynamoDB::Model::AttributeValue> attributeValues;
    attributeValues.emplace(":valueToMatch", partitionValue);

    request.SetExpressionAttributeValues(attributeValues);

    bool result = true;

    // "exclusiveStartKey" is used for pagination.
    Aws::Map<Aws::String, Aws::DynamoDB::Model::AttributeValue>
 exclusiveStartKey;
    do {
        if (!exclusiveStartKey.empty()) {
            request.SetExclusiveStartKey(exclusiveStartKey);
            exclusiveStartKey.clear();
        }
        // Perform Query operation.
```

```
        const Aws::DynamoDB::Model::QueryOutcome &outcome =
 dynamoClient.Query(request);
        if (outcome.IsSuccess()) {
            // Reference the retrieved items.
            const Aws::Vector<Aws::Map<Aws::String,
 Aws::DynamoDB::Model::AttributeValue>> &items = outcome.GetResult().GetItems();
            if (!items.empty()) {
                std::cout << "Number of items retrieved from Query: " <<
 items.size()
                          << std::endl;
                // Iterate each item and print.
                for (const auto &item: items) {
                    std::cout
                            <<
 "*************************************************"
                            << std::endl;
                    // Output each retrieved field and its value.
                    for (const auto &i: item)
                        std::cout << i.first << ": " << i.second.GetS() <<
 std::endl;
                }
            }
            else {
                std::cout << "No item found in table: " << tableName <<
 std::endl;
            }

            exclusiveStartKey = outcome.GetResult().GetLastEvaluatedKey();
        }
        else {
            std::cerr << "Failed to Query items: " <<
 outcome.GetError().GetMessage();
            result = false;
            break;
        }
    } while (!exclusiveStartKey.empty());

    return result;
}
```

- For API details, see Query in *AWS SDK for C++ API Reference*.

CLI

AWS CLI

Example 1: To query a table

The following `query` example queries items in the `MusicCollection` table. The table has a hash-and-range primary key (`Artist` and `SongTitle`), but this query only specifies the hash key value. It returns song titles by the artist named "No One You Know".

```
aws dynamodb query \
    --table-name MusicCollection \
    --projection-expression "SongTitle" \
    --key-condition-expression "Artist = :v1" \
    --expression-attribute-values file://expression-attributes.json \
    --return-consumed-capacity TOTAL
```

Contents of `expression-attributes.json`:

```
{
    ":v1": {"S": "No One You Know"}
}
```

Output:

```
{
    "Items": [
        {
            "SongTitle": {
                "S": "Call Me Today"
            },
            "SongTitle": {
                "S": "Scared of My Shadow"
            }
        }
    ],
    "Count": 2,
    "ScannedCount": 2,
    "ConsumedCapacity": {
        "TableName": "MusicCollection",
        "CapacityUnits": 0.5
```

```
    }
}
```

For more information, see Working with Queries in DynamoDB in the *Amazon DynamoDB Developer Guide*.

Example 2: To query a table using strongly consistent reads and traverse the index in descending order

The following example performs the same query as the first example, but returns results in reverse order and uses strongly consistent reads.

```
aws dynamodb query \
    --table-name MusicCollection \
    --projection-expression "SongTitle" \
    --key-condition-expression "Artist = :v1" \
    --expression-attribute-values file://expression-attributes.json \
    --consistent-read \
    --no-scan-index-forward \
    --return-consumed-capacity TOTAL
```

Contents of `expression-attributes.json`:

```
{
    ":v1": {"S": "No One You Know"}
}
```

Output:

```
{
    "Items": [
        {
            "SongTitle": {
                "S": "Scared of My Shadow"
            }
        },
        {
            "SongTitle": {
                "S": "Call Me Today"
            }
        }
```

```
    ],
    "Count": 2,
    "ScannedCount": 2,
    "ConsumedCapacity": {
        "TableName": "MusicCollection",
        "CapacityUnits": 1.0
    }
}
```

For more information, see Working with Queries in DynamoDB in the *Amazon DynamoDB Developer Guide*.

Example 3: To filter out specific results

The following example queries the `MusicCollection` but excludes results with specific values in the `AlbumTitle` attribute. Note that this does not affect the `ScannedCount` or `ConsumedCapacity`, because the filter is applied after the items have been read.

```
aws dynamodb query \
    --table-name MusicCollection \
    --key-condition-expression "#n1 = :v1" \
    --filter-expression "NOT (#n2 IN (:v2, :v3))" \
    --expression-attribute-names file://names.json \
    --expression-attribute-values file://values.json \
    --return-consumed-capacity TOTAL
```

Contents of `values.json`:

```
{
    ":v1": {"S": "No One You Know"},
    ":v2": {"S": "Blue Sky Blues"},
    ":v3": {"S": "Greatest Hits"}
}
```

Contents of `names.json`:

```
{
    "#n1": "Artist",
    "#n2": "AlbumTitle"
}
```

Output:

```
{
    "Items": [
        {
            "AlbumTitle": {
                "S": "Somewhat Famous"
            },
            "Artist": {
                "S": "No One You Know"
            },
            "SongTitle": {
                "S": "Call Me Today"
            }
        }
    ],
    "Count": 1,
    "ScannedCount": 2,
    "ConsumedCapacity": {
        "TableName": "MusicCollection",
        "CapacityUnits": 0.5
    }
}
```

For more information, see Working with Queries in DynamoDB in the *Amazon DynamoDB Developer Guide*.

Example 4: To retrieve only an item count

The following example retrieves a count of items matching the query, but does not retrieve any of the items themselves.

```
aws dynamodb query \
    --table-name MusicCollection \
    --select COUNT \
    --key-condition-expression "Artist = :v1" \
    --expression-attribute-values file://expression-attributes.json
```

Contents of `expression-attributes.json`:

```
{
```

```
    ":v1": {"S": "No One You Know"}
}
```

Output:

```
{
    "Count": 2,
    "ScannedCount": 2,
    "ConsumedCapacity": null
}
```

For more information, see Working with Queries in DynamoDB in the *Amazon DynamoDB Developer Guide*.

Example 5: To query an index

The following example queries the local secondary index `AlbumTitleIndex`. The query returns all attributes from the base table that have been projected into the local secondary index. Note that when querying a local secondary index or global secondary index, you must also provide the name of the base table using the `table-name` parameter.

```
aws dynamodb query \
    --table-name MusicCollection \
    --index-name AlbumTitleIndex \
    --key-condition-expression "Artist = :v1" \
    --expression-attribute-values file://expression-attributes.json \
    --select ALL_PROJECTED_ATTRIBUTES \
    --return-consumed-capacity INDEXES
```

Contents of `expression-attributes.json`:

```
{
    ":v1": {"S": "No One You Know"}
}
```

Output:

```
{
    "Items": [
```

```
        {
            "AlbumTitle": {
                "S": "Blue Sky Blues"
            },
            "Artist": {
                "S": "No One You Know"
            },
            "SongTitle": {
                "S": "Scared of My Shadow"
            }
        },
        {
            "AlbumTitle": {
                "S": "Somewhat Famous"
            },
            "Artist": {
                "S": "No One You Know"
            },
            "SongTitle": {
                "S": "Call Me Today"
            }
        }
    ],
    "Count": 2,
    "ScannedCount": 2,
    "ConsumedCapacity": {
        "TableName": "MusicCollection",
        "CapacityUnits": 0.5,
        "Table": {
            "CapacityUnits": 0.0
        },
        "LocalSecondaryIndexes": {
            "AlbumTitleIndex": {
                "CapacityUnits": 0.5
            }
        }
    }
}
```

For more information, see Working with Queries in DynamoDB in the *Amazon DynamoDB Developer Guide*.

- For API details, see Query in *AWS CLI Command Reference*.

Go

SDK for Go V2

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import (
 "context"
 "errors"
 "log"
 "time"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/attributevalue"
 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/expression"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb/types"
)

// TableBasics encapsulates the Amazon DynamoDB service actions used in the
 examples.
// It contains a DynamoDB service client that is used to act on the specified
 table.
type TableBasics struct {
 DynamoDbClient *dynamodb.Client
 TableName      string
}



// Query gets all movies in the DynamoDB table that were released in the
 specified year.
// The function uses the `expression` package to build the key condition
 expression
// that is used in the query.
func (basics TableBasics) Query(ctx context.Context, releaseYear int) ([]Movie,
 error) {
```

```
 var err error
 var response *dynamodb.QueryOutput
 var movies []Movie
 keyEx := expression.Key("year").Equal(expression.Value(releaseYear))
 expr, err := expression.NewBuilder().WithKeyCondition(keyEx).Build()
 if err != nil {
  log.Printf("Couldn't build expression for query. Here's why: %v\n", err)
 } else {
  queryPaginator := dynamodb.NewQueryPaginator(basics.DynamoDbClient,
 &dynamodb.QueryInput{
   TableName:                 aws.String(basics.TableName),
   ExpressionAttributeNames:  expr.Names(),
   ExpressionAttributeValues: expr.Values(),
   KeyConditionExpression:    expr.KeyCondition(),
  })
  for queryPaginator.HasMorePages() {
   response, err = queryPaginator.NextPage(ctx)
   if err != nil {
    log.Printf("Couldn't query for movies released in %v. Here's why: %v\n",
 releaseYear, err)
    break
   } else {
    var moviePage []Movie
    err = attributevalue.UnmarshalListOfMaps(response.Items, &moviePage)
    if err != nil {
     log.Printf("Couldn't unmarshal query response. Here's why: %v\n", err)
     break
    } else {
     movies = append(movies, moviePage...)
    }
   }
  }
 }
 return movies, err
}
```

Define a Movie struct that is used in this example.

```
import (
 "archive/zip"
```

```
 "bytes"
 "encoding/json"
 "fmt"
 "io"
 "log"
 "net/http"

 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/attributevalue"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb/types"
)

// Movie encapsulates data about a movie. Title and Year are the composite
 primary key
// of the movie in Amazon DynamoDB. Title is the sort key, Year is the partition
 key,
// and Info is additional data.
type Movie struct {
 Title string                `dynamodbav:"title"`
 Year  int                   `dynamodbav:"year"`
 Info  map[string]interface{} `dynamodbav:"info"`
}

// GetKey returns the composite primary key of the movie in a format that can be
// sent to DynamoDB.
func (movie Movie) GetKey() map[string]types.AttributeValue {
 title, err := attributevalue.Marshal(movie.Title)
 if err != nil {
  panic(err)
 }
 year, err := attributevalue.Marshal(movie.Year)
 if err != nil {
  panic(err)
 }
 return map[string]types.AttributeValue{"title": title, "year": year}
}

// String returns the title, year, rating, and plot of a movie, formatted for the
 example.
func (movie Movie) String() string {
 return fmt.Sprintf("%v\n\tReleased: %v\n\tRating: %v\n\tPlot: %v\n",
  movie.Title, movie.Year, movie.Info["rating"], movie.Info["plot"])
}
```

- For API details, see Query in *AWS SDK for Go API Reference*.

Java

SDK for Java 2.x

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Queries a table by using DynamoDbClient.

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.QueryRequest;
import software.amazon.awssdk.services.dynamodb.model.QueryResponse;
import java.util.HashMap;

/**
 * Before running this Java V2 code example, set up your development
 * environment, including your credentials.
 *
 * For more information, see the following documentation topic:
 *
 * https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-
started.html
 *
 * To query items from an Amazon DynamoDB table using the AWS SDK for Java V2,
 * its better practice to use the
 * Enhanced Client. See the EnhancedQueryRecords example.
 */
public class Query {
    public static void main(String[] args) {
        final String usage = """

                Usage:
```

```
                    <tableName> <partitionKeyName> <partitionKeyVal>

                Where:
                    tableName - The Amazon DynamoDB table to put the item in (for
 example, Music3).
                    partitionKeyName - The partition key name of the Amazon
 DynamoDB table (for example, Artist).
                    partitionKeyVal - The value of the partition key that should
 match (for example, Famous Band).
                """;

        if (args.length != 3) {
            System.out.println(usage);
            System.exit(1);
        }

        String tableName = args[0];
        String partitionKeyName = args[1];
        String partitionKeyVal = args[2];

        // For more information about an alias, see:
        // https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/
Expressions.ExpressionAttributeNames.html
        String partitionAlias = "#a";

        System.out.format("Querying %s", tableName);
        System.out.println("");
        Region region = Region.US_EAST_1;
        DynamoDbClient ddb = DynamoDbClient.builder()
                .region(region)
                .build();

        int count = queryTable(ddb, tableName, partitionKeyName, partitionKeyVal,
 partitionAlias);
        System.out.println("There were " + count + "  record(s) returned");
        ddb.close();
    }

    public static int queryTable(DynamoDbClient ddb, String tableName, String
 partitionKeyName, String partitionKeyVal,
            String partitionAlias) {
        // Set up an alias for the partition key name in case it's a reserved
 word.
        HashMap<String, String> attrNameAlias = new HashMap<String, String>();
```

```
        attrNameAlias.put(partitionAlias, partitionKeyName);

        // Set up mapping of the partition name with the value.
        HashMap<String, AttributeValue> attrValues = new HashMap<>();
        attrValues.put(":" + partitionKeyName, AttributeValue.builder()
                .s(partitionKeyVal)
                .build());

        QueryRequest queryReq = QueryRequest.builder()
                .tableName(tableName)
                .keyConditionExpression(partitionAlias + " = :" +
 partitionKeyName)
                .expressionAttributeNames(attrNameAlias)
                .expressionAttributeValues(attrValues)
                .build();

        try {
            QueryResponse response = ddb.query(queryReq);
            return response.count();

        } catch (DynamoDbException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
        return -1;
    }
}
```

Queries a table by using `DynamoDbClient` and a secondary index.

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.QueryRequest;
import software.amazon.awssdk.services.dynamodb.model.QueryResponse;
import java.util.HashMap;
import java.util.Map;

/**
 * Before running this Java V2 code example, set up your development
 * environment, including your credentials.
```

```
 *
 * For more information, see the following documentation topic:
 *
 * https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-
started.html
 *
 * Create the Movies table by running the Scenario example and loading the Movie
 * data from the JSON file. Next create a secondary
 * index for the Movies table that uses only the year column. Name the index
 * **year-index**. For more information, see:
 *
 * https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/GSI.html
 */
public class QueryItemsUsingIndex {
    public static void main(String[] args) {
        String tableName = "Movies";
        Region region = Region.US_EAST_1;
        DynamoDbClient ddb = DynamoDbClient.builder()
                .region(region)
                .build();

        queryIndex(ddb, tableName);
        ddb.close();
    }

    public static void queryIndex(DynamoDbClient ddb, String tableName) {
        try {
            Map<String, String> expressionAttributesNames = new HashMap<>();
            expressionAttributesNames.put("#year", "year");
            Map<String, AttributeValue> expressionAttributeValues = new
 HashMap<>();
            expressionAttributeValues.put(":yearValue",
 AttributeValue.builder().n("2013").build());

            QueryRequest request = QueryRequest.builder()
                    .tableName(tableName)
                    .indexName("year-index")
                    .keyConditionExpression("#year = :yearValue")
                    .expressionAttributeNames(expressionAttributesNames)
                    .expressionAttributeValues(expressionAttributeValues)
                    .build();

            System.out.println("=== Movie Titles ===");
            QueryResponse response = ddb.query(request);
```

```
            response.items()
                    .forEach(movie ->
 System.out.println(movie.get("title").s()));

        } catch (DynamoDbException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }
}
```

- For API details, see Query in *AWS SDK for Java 2.x API Reference*.

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

This example uses the document client to simplify working with items in DynamoDB. For API details see QueryCommand.

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
import { QueryCommand, DynamoDBDocumentClient } from "@aws-sdk/lib-dynamodb";

const client = new DynamoDBClient({});
const docClient = DynamoDBDocumentClient.from(client);

export const main = async () => {
  const command = new QueryCommand({
    TableName: "CoffeeCrop",
    KeyConditionExpression:
      "OriginCountry = :originCountry AND RoastDate > :roastDate",
    ExpressionAttributeValues: {
      ":originCountry": "Ethiopia",
      ":roastDate": "2023-05-01",
    },
```

```
    ConsistentRead: true,
  });

  const response = await docClient.send(command);
  console.log(response);
  return response;
};
```

- For more information, see AWS SDK for JavaScript Developer Guide.
- For API details, see Query in *AWS SDK for JavaScript API Reference*.

SDK for JavaScript (v2)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create DynamoDB document client
var docClient = new AWS.DynamoDB.DocumentClient({ apiVersion: "2012-08-10" });

var params = {
  ExpressionAttributeValues: {
    ":s": 2,
    ":e": 9,
    ":topic": "PHRASE",
  },
  KeyConditionExpression: "Season = :s and Episode > :e",
  FilterExpression: "contains (Subtitle, :topic)",
  TableName: "EPISODES_TABLE",
};

docClient.query(params, function (err, data) {
  if (err) {
    console.log("Error", err);
```

```
  } else {
    console.log("Success", data.Items);
  }
});
```

- For more information, see AWS SDK for JavaScript Developer Guide.
- For API details, see Query in *AWS SDK for JavaScript API Reference*.

Kotlin

SDK for Kotlin

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
suspend fun queryDynTable(
    tableNameVal: String,
    partitionKeyName: String,
    partitionKeyVal: String,
    partitionAlias: String,
): Int {
    val attrNameAlias = mutableMapOf<String, String>()
    attrNameAlias[partitionAlias] = partitionKeyName

    // Set up mapping of the partition name with the value.
    val attrValues = mutableMapOf<String, AttributeValue>()
    attrValues[":$partitionKeyName"] = AttributeValue.S(partitionKeyVal)

    val request =
        QueryRequest {
            tableName = tableNameVal
            keyConditionExpression = "$partitionAlias = :$partitionKeyName"
            expressionAttributeNames = attrNameAlias
            this.expressionAttributeValues = attrValues
        }

    DynamoDbClient.fromEnvironment { region = "us-east-1" }.use { ddb ->
```

```
        val response = ddb.query(request)
        return response.count
    }
}
```

- For API details, see Query in *AWS SDK for Kotlin API reference*.

PHP

SDK for PHP

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
        $birthKey = [
            'Key' => [
                'year' => [
                    'N' => "$birthYear",
                ],
            ],
        ];
        $result = $service->query($tableName, $birthKey);

    public function query(string $tableName, $key)
    {
        $expressionAttributeValues = [];
        $expressionAttributeNames = [];
        $keyConditionExpression = "";
        $index = 1;
        foreach ($key as $name => $value) {
            $keyConditionExpression .= "#" . array_key_first($value) . " = :v
$index,";
            $expressionAttributeNames["#" . array_key_first($value)] =
 array_key_first($value);
            $hold = array_pop($value);
            $expressionAttributeValues[":v$index"] = [
                array_key_first($hold) => array_pop($hold),
            ];
```

```
        }
        $keyConditionExpression = substr($keyConditionExpression, 0, -1);
        $query = [
            'ExpressionAttributeValues' => $expressionAttributeValues,
            'ExpressionAttributeNames' => $expressionAttributeNames,
            'KeyConditionExpression' => $keyConditionExpression,
            'TableName' => $tableName,
        ];
        return $this->dynamoDbClient->query($query);
    }
```

- For API details, see Query in *AWS SDK for PHP API Reference*.

PowerShell

Tools for PowerShell V4

Example 1: Invokes a query that returns DynamoDB items with the specified SongTitle and Artist.

```
$invokeDDBQuery = @{
    TableName = 'Music'
    KeyConditionExpression = ' SongTitle = :SongTitle and Artist = :Artist'
    ExpressionAttributeValues = @{
        ':SongTitle' = 'Somewhere Down The Road'
        ':Artist' = 'No One You Know'
    } | ConvertTo-DDBItem
}
Invoke-DDBQuery @invokeDDBQuery | ConvertFrom-DDBItem
```

Output:

```
Name                           Value
----                           -----
Genre                          Country
Artist                         No One You Know
Price                          1.94
CriticRating                   9
SongTitle                      Somewhere Down The Road
AlbumTitle                     Somewhat Famous
```

- For API details, see Query in *AWS Tools for PowerShell Cmdlet Reference (V4)*.

Tools for PowerShell V5

Example 1: Invokes a query that returns DynamoDB items with the specified SongTitle and Artist.

```
$invokeDDBQuery = @{
    TableName = 'Music'
    KeyConditionExpression = ' SongTitle = :SongTitle and Artist = :Artist'
    ExpressionAttributeValues = @{
        ':SongTitle' = 'Somewhere Down The Road'
        ':Artist' = 'No One You Know'
    } | ConvertTo-DDBItem
}
Invoke-DDBQuery @invokeDDBQuery | ConvertFrom-DDBItem
```

Output:

```
Name                           Value
----                           -----
Genre                          Country
Artist                         No One You Know
Price                          1.94
CriticRating                   9
SongTitle                      Somewhere Down The Road
AlbumTitle                     Somewhat Famous
```

- For API details, see Query in *AWS Tools for PowerShell Cmdlet Reference (V5)*.

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Query items by using a key condition expression.

```
class Movies:
    """Encapsulates an Amazon DynamoDB table of movie data.

    Example data structure for a movie record in this table:
        {
            "year": 1999,
            "title": "For Love of the Game",
            "info": {
                "directors": ["Sam Raimi"],
                "release_date": "1999-09-15T00:00:00Z",
                "rating": 6.3,
                "plot": "A washed up pitcher flashes through his career.",
                "rank": 4987,
                "running_time_secs": 8220,
                "actors": [
                    "Kevin Costner",
                    "Kelly Preston",
                    "John C. Reilly"
                ]
            }
        }
    """

    def __init__(self, dyn_resource):
        """
        :param dyn_resource: A Boto3 DynamoDB resource.
        """
        self.dyn_resource = dyn_resource
        # The table variable is set during the scenario in the call to
        # 'exists' if the table exists. Otherwise, it is set by 'create_table'.
        self.table = None


    def query_movies(self, year):
        """
        Queries for movies that were released in the specified year.

        :param year: The year to query.
        :return: The list of movies that were released in the specified year.
        """
        try:
            response =
 self.table.query(KeyConditionExpression=Key("year").eq(year))
```

```
        except ClientError as err:
            logger.error(
                "Couldn't query for movies released in %s. Here's why: %s: %s",
                year,
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
        else:
            return response["Items"]
```

Query items and project them to return a subset of data.

```
class UpdateQueryWrapper:
    def __init__(self, table):
        self.table = table


    def query_and_project_movies(self, year, title_bounds):
        """
        Query for movies that were released in a specified year and that have
 titles
        that start within a range of letters. A projection expression is used
        to return a subset of data for each movie.

        :param year: The release year to query.
        :param title_bounds: The range of starting letters to query.
        :return: The list of movies.
        """
        try:
            response = self.table.query(
                ProjectionExpression="#yr, title, info.genres, info.actors[0]",
                ExpressionAttributeNames={"#yr": "year"},
                KeyConditionExpression=(
                    Key("year").eq(year)
                    & Key("title").between(
                        title_bounds["first"], title_bounds["second"]
                    )
                ),
            )
        except ClientError as err:
```

```
            if err.response["Error"]["Code"] == "ValidationException":
                logger.warning(
                    "There's a validation error. Here's the message: %s: %s",
                    err.response["Error"]["Code"],
                    err.response["Error"]["Message"],
                )
            else:
                logger.error(
                    "Couldn't query for movies. Here's why: %s: %s",
                    err.response["Error"]["Code"],
                    err.response["Error"]["Message"],
                )
                raise
        else:
            return response["Items"]
```

- For API details, see Query in *AWS SDK for Python (Boto3) API Reference*.

Ruby

SDK for Ruby

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
class DynamoDBBasics
  attr_reader :dynamo_resource, :table

  def initialize(table_name)
    client = Aws::DynamoDB::Client.new(region: 'us-east-1')
    @dynamo_resource = Aws::DynamoDB::Resource.new(client: client)
    @table = @dynamo_resource.table(table_name)
  end

  # Queries for movies that were released in the specified year.
  #
  # @param year [Integer] The year to query.
```

```
  # @return [Array] The list of movies that were released in the specified year.
  def query_items(year)
    response = @table.query(
      key_condition_expression: '#yr = :year',
      expression_attribute_names: { '#yr' => 'year' },
      expression_attribute_values: { ':year' => year }
    )
  rescue Aws::DynamoDB::Errors::ServiceError => e
    puts("Couldn't query for movies released in #{year}. Here's why:")
    puts("\t#{e.code}: #{e.message}")
    raise
  else
    response.items
  end
```

- For API details, see Query in *AWS SDK for Ruby API Reference*.

Rust

SDK for Rust

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Find the movies made in the specified year.

```
pub async fn movies_in_year(
    client: &Client,
    table_name: &str,
    year: u16,
) -> Result<Vec<Movie>, MovieError> {
    let results = client
        .query()
        .table_name(table_name)
        .key_condition_expression("#yr = :yyyy")
        .expression_attribute_names("#yr", "year")
        .expression_attribute_values(":yyyy",
 AttributeValue::N(year.to_string()))
```

```
        .send()
        .await?;

    if let Some(items) = results.items {
        let movies = items.iter().map(|v| v.into()).collect();
        Ok(movies)
    } else {
        Ok(vec![])
    }
}
```

- For API details, see Query in *AWS SDK for Rust API reference*.

SAP ABAP

SDK for SAP ABAP

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    TRY.
        " Query movies for a given year .
        DATA(lt_attributelist) = VALUE /aws1/
cl_dynattributevalue=>tt_attributevaluelist(
            ( NEW /aws1/cl_dynattributevalue( iv_n = |{ iv_year }| ) ) ).
        DATA(lt_key_conditions) = VALUE /aws1/cl_dyncondition=>tt_keyconditions(
          ( VALUE /aws1/cl_dyncondition=>ts_keyconditions_maprow(
          key = 'year'
          value = NEW /aws1/cl_dyncondition(
          it_attributevaluelist = lt_attributelist
          iv_comparisonoperator = |EQ|
          ) ) ) ).
        oo_result = lo_dyn->query(
          iv_tablename = iv_table_name
          it_keyconditions = lt_key_conditions ).
        DATA(lt_items) = oo_result->get_items( ).
        "You can loop over the results to get item attributes.
```

```
        LOOP AT lt_items INTO DATA(lt_item).
          DATA(lo_title) = lt_item[ key = 'title' ]-value.
          DATA(lo_year) = lt_item[ key = 'year' ]-value.
        ENDLOOP.
        DATA(lv_count) = oo_result->get_count( ).
        MESSAGE 'Item count is: ' && lv_count TYPE 'I'.
      CATCH /aws1/cx_dynresourcenotfoundex.
        MESSAGE 'The table or index does not exist' TYPE 'E'.
    ENDTRY.
```

- For API details, see Query in *AWS SDK for SAP ABAP API reference*.

Swift

SDK for Swift

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import AWSDynamoDB


    /// Get all the movies released in the specified year.
    ///
    /// - Parameter year: The release year of the movies to return.
    ///
    /// - Returns: An array of `Movie` objects describing each matching movie.
    ///
    func getMovies(fromYear year: Int) async throws -> [Movie] {
        do {
            guard let client = self.ddbClient else {
                throw MoviesError.UninitializedClient
            }

            let input = QueryInput(
                expressionAttributeNames: [
                    "#y": "year"
                ],
```

```
                expressionAttributeValues: [
                    ":y": .n(String(year))
                ],
                keyConditionExpression: "#y = :y",
                tableName: self.tableName
            )
            // Use "Paginated" to get all the movies.
            // This lets the SDK handle the 'lastEvaluatedKey' property in
"QueryOutput".

            let pages = client.queryPaginated(input: input)

            var movieList: [Movie] = []
            for try await page in pages {
                guard let items = page.items else {
                    print("Error: no items returned.")
                    continue
                }

                // Convert the found movies into `Movie` objects and return an
array
                // of them.

                for item in items {
                    let movie = try Movie(withItem: item)
                    movieList.append(movie)
                }
            }
            return movieList
        } catch {
            print("ERROR: getMovies:", dump(error))
            throw error
        }
    }
```

- For API details, see Query in *AWS SDK for Swift API reference*.

For a complete list of AWS SDK developer guides and code examples, see Using DynamoDB with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use Scan with an AWS SDK or CLI

The following code examples show how to use Scan.

Action examples are code excerpts from larger programs and must be run in context. You can see this action in context in the following code examples:

- Learn the basics
- Accelerate reads with DAX
- Compare multiple values with a single attribute

.NET

SDK for .NET (v4)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    /// <summary>
    /// Scans the table for movies released between the specified years.
    /// </summary>
    /// <param name="tableName">The name of the table to scan.</param>
    /// <param name="startYear">The starting year for the range.</param>
    /// <param name="endYear">The ending year for the range.</param>
    /// <returns>The number of movies found in the specified year range.</
returns>
    public async Task<int> ScanTableAsync(
        string tableName,
        int startYear,
        int endYear)
    {
        try
        {
            var request = new ScanRequest
            {
                TableName = tableName,
                ExpressionAttributeNames = new Dictionary<string, string>
                {
```

```
                    { "#yr", "year" },
                },
                ExpressionAttributeValues = new Dictionary<string,
AttributeValue>
                {
                    { ":y_a", new AttributeValue { N = startYear.ToString() } },
                    { ":y_z", new AttributeValue { N = endYear.ToString() } },
                },
                FilterExpression = "#yr between :y_a and :y_z",
                ProjectionExpression = "#yr, title, info.actors[0],
info.directors, info.running_time_secs",
                Limit = 10 // Set a limit to demonstrate using the
LastEvaluatedKey.
            };

            // Keep track of how many movies were found.
            int foundCount = 0;

            var response = new ScanResponse();
            do
            {
                response = await _amazonDynamoDB.ScanAsync(request);
                foundCount += response.Items.Count;
                response.Items.ForEach(i => DisplayItem(i));
                request.ExclusiveStartKey = response.LastEvaluatedKey;
            }
            while (response?.LastEvaluatedKey?.Count > 0);
            return foundCount;
        }
        catch (ResourceNotFoundException ex)
        {
            Console.WriteLine($"Table {tableName} was not found. {ex.Message}");
            return 0;
        }
        catch (AmazonDynamoDBException ex)
        {
            Console.WriteLine($"An Amazon DynamoDB error occurred while scanning
table. {ex.Message}");
            throw;
        }
        catch (Exception ex)
        {
            Console.WriteLine($"An error occurred while scanning table.
{ex.Message}");
```

```
            throw;
        }
    }
```

- For API details, see Scan in *AWS SDK for .NET API Reference*.

Bash

AWS CLI with Bash script

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
###############################################################################
# function dynamodb_scan
#
# This function scans a DynamoDB table.
#
# Parameters:
#       -n table_name  -- The name of the table.
#       -f filter_expression  -- The filter expression.
#       -a expression_attribute_names -- Path to JSON file containing the
 expression attribute names.
#       -v expression_attribute_values -- Path to JSON file containing the
 expression attribute values.
#       [-p projection_expression]  -- Optional projection expression.
#
#  Returns:
#       The items as json output.
#  And:
#       0 - If successful.
#       1 - If it fails.
###############################################################################
function dynamodb_scan() {
  local table_name filter_expression expression_attribute_names
 expression_attribute_values projection_expression response
  local option OPTARG # Required to use getopts command in a function.
```

```
 # ######################################
 # Function usage explanation
 ######################################
 function usage() {
   echo "function dynamodb_scan"
   echo "Scan a DynamoDB table."
   echo " -n table_name  -- The name of the table."
   echo " -f filter_expression  -- The filter expression."
   echo " -a expression_attribute_names -- Path to JSON file containing the
expression attribute names."
   echo " -v expression_attribute_values -- Path to JSON file containing the
expression attribute values."
   echo " [-p projection_expression]  -- Optional projection expression."
   echo ""
 }

 while getopts "n:f:a:v:p:h" option; do
   case "${option}" in
     n) table_name="${OPTARG}" ;;
     f) filter_expression="${OPTARG}" ;;
     a) expression_attribute_names="${OPTARG}" ;;
     v) expression_attribute_values="${OPTARG}" ;;
     p) projection_expression="${OPTARG}" ;;
     h)
       usage
       return 0
       ;;
     \?)
       echo "Invalid parameter"
       usage
       return 1
       ;;
   esac
 done
 export OPTIND=1

 if [[ -z "$table_name" ]]; then
   errecho "ERROR: You must provide a table name with the -n parameter."
   usage
   return 1
 fi

 if [[ -z "$filter_expression" ]]; then
```

```
    errecho "ERROR: You must provide a filter expression with the -f parameter."
    usage
    return 1
  fi

  if [[ -z "$expression_attribute_names" ]]; then
    errecho "ERROR: You must provide expression attribute names with the -a
parameter."
    usage
    return 1
  fi

  if [[ -z "$expression_attribute_values" ]]; then
    errecho "ERROR: You must provide expression attribute values with the -v
parameter."
    usage
    return 1
  fi

  if [[ -z "$projection_expression" ]]; then
    response=$(aws dynamodb scan \
      --table-name "$table_name" \
      --filter-expression "$filter_expression" \
      --expression-attribute-names file://"$expression_attribute_names" \
      --expression-attribute-values file://"$expression_attribute_values")
  else
    response=$(aws dynamodb scan \
      --table-name "$table_name" \
      --filter-expression "$filter_expression" \
      --expression-attribute-names file://"$expression_attribute_names" \
      --expression-attribute-values file://"$expression_attribute_values" \
      --projection-expression "$projection_expression")
  fi

  local error_code=${?}

  if [[ $error_code -ne 0 ]]; then
    aws_cli_error_log $error_code
    errecho "ERROR: AWS reports scan operation failed.$response"
    return 1
  fi

  echo "$response"
```

```
  return 0
}
```

The utility functions used in this example.

```
###############################################################################
# function errecho
#
# This function outputs everything sent to it to STDERR (standard error output).
###############################################################################
function errecho() {
  printf "%s\n" "$*" 1>&2
}

###############################################################################
# function aws_cli_error_log()
#
# This function is used to log the error messages from the AWS CLI.
#
# See https://docs.aws.amazon.com/cli/latest/topic/return-codes.html#cli-aws-
help-return-codes.
#
# The function expects the following argument:
#         $1 - The error code returned by the AWS CLI.
#
#  Returns:
#          0: - Success.
#
###############################################################################
function aws_cli_error_log() {
  local err_code=$1
  errecho "Error code : $err_code"
  if [ "$err_code" == 1 ]; then
    errecho "  One or more S3 transfers failed."
  elif [ "$err_code" == 2 ]; then
    errecho "  Command line failed to parse."
  elif [ "$err_code" == 130 ]; then
    errecho "  Process received SIGINT."
  elif [ "$err_code" == 252 ]; then
    errecho "  Command syntax invalid."
  elif [ "$err_code" == 253 ]; then
    errecho "  The system environment or configuration was invalid."
```

```
  elif [ "$err_code" == 254 ]; then
    errecho "  The service returned an error."
  elif [ "$err_code" == 255 ]; then
    errecho "  255 is a catch-all error."
  fi

  return 0
}
```

- For API details, see Scan in *AWS CLI Command Reference*.

C++

SDK for C++

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
//! Scan an Amazon DynamoDB table.
/*!
  \sa scanTable()
  \param tableName: Name for the DynamoDB table.
  \param projectionExpression: An optional projection expression, ignored if
 empty.
  \param clientConfiguration: AWS client configuration.
  \return bool: Function succeeded.
 */

bool AwsDoc::DynamoDB::scanTable(const Aws::String &tableName,
                                 const Aws::String &projectionExpression,
                                 const Aws::Client::ClientConfiguration
 &clientConfiguration) {
    Aws::DynamoDB::DynamoDBClient dynamoClient(clientConfiguration);
    Aws::DynamoDB::Model::ScanRequest request;
    request.SetTableName(tableName);

    if (!projectionExpression.empty())
        request.SetProjectionExpression(projectionExpression);
```

```
   Aws::Vector<Aws::Map<Aws::String, Aws::DynamoDB::Model::AttributeValue>>
all_items;
   Aws::Map<Aws::String, Aws::DynamoDB::Model::AttributeValue>
last_evaluated_key; // Used for pagination;
   do {
       if (!last_evaluated_key.empty()) {
           request.SetExclusiveStartKey(last_evaluated_key);
       }
       const Aws::DynamoDB::Model::ScanOutcome &outcome =
dynamoClient.Scan(request);
       if (outcome.IsSuccess()) {
           // Reference the retrieved items.
           const Aws::Vector<Aws::Map<Aws::String,
Aws::DynamoDB::Model::AttributeValue>> &items = outcome.GetResult().GetItems();
           all_items.insert(all_items.end(), items.begin(), items.end());

           last_evaluated_key = outcome.GetResult().GetLastEvaluatedKey();
       }
       else {
           std::cerr << "Failed to Scan items: " <<
outcome.GetError().GetMessage()
                     << std::endl;
           return false;
       }

   } while (!last_evaluated_key.empty());

   if (!all_items.empty()) {
       std::cout << "Number of items retrieved from scan: " << all_items.size()
                 << std::endl;
       // Iterate each item and print.
       for (const Aws::Map<Aws::String, Aws::DynamoDB::Model::AttributeValue>
&itemMap: all_items) {
           std::cout << "*************************************************"
                     << std::endl;
           // Output each retrieved field and its value.
           for (const auto &itemEntry: itemMap)
               std::cout << itemEntry.first << ": " << itemEntry.second.GetS()
                         << std::endl;
       }
   }

   else {
```

```
        std::cout << "No items found in table: " << tableName << std::endl;
    }

    return true;
}
```

- For API details, see Scan in *AWS SDK for C++ API Reference*.

CLI

AWS CLI

To scan a table

The following `scan` example scans the entire `MusicCollection` table, and then narrows the results to songs by the artist "No One You Know". For each item, only the album title and song title are returned.

```
aws dynamodb scan \
    --table-name MusicCollection \
    --filter-expression "Artist = :a" \
    --projection-expression "#ST, #AT" \
    --expression-attribute-names file://expression-attribute-names.json \
    --expression-attribute-values file://expression-attribute-values.json
```

Contents of `expression-attribute-names.json`:

```
{
    "#ST": "SongTitle",
    "#AT":"AlbumTitle"
}
```

Contents of `expression-attribute-values.json`:

```
{
    ":a": {"S": "No One You Know"}
}
```

Output:

```
{
    "Count": 2,
    "Items": [
        {
            "SongTitle": {
                "S": "Call Me Today"
            },
            "AlbumTitle": {
                "S": "Somewhat Famous"
            }
        },
        {
            "SongTitle": {
                "S": "Scared of My Shadow"
            },
            "AlbumTitle": {
                "S": "Blue Sky Blues"
            }
        }
    ],
    "ScannedCount": 3,
    "ConsumedCapacity": null
}
```

For more information, see Working with Scans in DynamoDB in the *Amazon DynamoDB Developer Guide*.

- For API details, see Scan in *AWS CLI Command Reference*.

Go

SDK for Go V2

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import (
```

```
 "context"
 "errors"
 "log"
 "time"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/attributevalue"
 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/expression"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb/types"
)

// TableBasics encapsulates the Amazon DynamoDB service actions used in the
 examples.
// It contains a DynamoDB service client that is used to act on the specified
 table.
type TableBasics struct {
 DynamoDbClient *dynamodb.Client
 TableName      string
}



// Scan gets all movies in the DynamoDB table that were released in a range of
 years
// and projects them to return a reduced set of fields.
// The function uses the `expression` package to build the filter and projection
// expressions.
func (basics TableBasics) Scan(ctx context.Context, startYear int, endYear int)
 ([]Movie, error) {
 var movies []Movie
 var err error
 var response *dynamodb.ScanOutput
 filtEx := expression.Name("year").Between(expression.Value(startYear),
 expression.Value(endYear))
 projEx := expression.NamesList(
  expression.Name("year"), expression.Name("title"),
 expression.Name("info.rating"))
 expr, err :=
 expression.NewBuilder().WithFilter(filtEx).WithProjection(projEx).Build()
 if err != nil {
  log.Printf("Couldn't build expressions for scan. Here's why: %v\n", err)
 } else {
```

```
  scanPaginator := dynamodb.NewScanPaginator(basics.DynamoDbClient,
 &dynamodb.ScanInput{
   TableName:                 aws.String(basics.TableName),
   ExpressionAttributeNames:  expr.Names(),
   ExpressionAttributeValues: expr.Values(),
   FilterExpression:          expr.Filter(),
   ProjectionExpression:      expr.Projection(),
  })
  for scanPaginator.HasMorePages() {
   response, err = scanPaginator.NextPage(ctx)
   if err != nil {
    log.Printf("Couldn't scan for movies released between %v and %v. Here's why:
 %v\n",
     startYear, endYear, err)
    break
   } else {
    var moviePage []Movie
    err = attributevalue.UnmarshalListOfMaps(response.Items, &moviePage)
    if err != nil {
     log.Printf("Couldn't unmarshal query response. Here's why: %v\n", err)
     break
    } else {
     movies = append(movies, moviePage...)
    }
   }
  }
 }
 return movies, err
}
```

Define a Movie struct that is used in this example.

```
import (
 "archive/zip"
 "bytes"
 "encoding/json"
 "fmt"
 "io"
 "log"
 "net/http"
```

```
 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/attributevalue"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb/types"
)

// Movie encapsulates data about a movie. Title and Year are the composite
 primary key
// of the movie in Amazon DynamoDB. Title is the sort key, Year is the partition
 key,
// and Info is additional data.
type Movie struct {
 Title string                 `dynamodbav:"title"`
 Year  int                    `dynamodbav:"year"`
 Info  map[string]interface{} `dynamodbav:"info"`
}

// GetKey returns the composite primary key of the movie in a format that can be
// sent to DynamoDB.
func (movie Movie) GetKey() map[string]types.AttributeValue {
 title, err := attributevalue.Marshal(movie.Title)
 if err != nil {
  panic(err)
 }
 year, err := attributevalue.Marshal(movie.Year)
 if err != nil {
  panic(err)
 }
 return map[string]types.AttributeValue{"title": title, "year": year}
}

// String returns the title, year, rating, and plot of a movie, formatted for the
 example.
func (movie Movie) String() string {
 return fmt.Sprintf("%v\n\tReleased: %v\n\tRating: %v\n\tPlot: %v\n",
  movie.Title, movie.Year, movie.Info["rating"], movie.Info["plot"])
}
```

- For API details, see Scan in *AWS SDK for Go API Reference*.

Java

SDK for Java 2.x

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Scans an Amazon DynamoDB table using DynamoDbClient.

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.ScanRequest;
import software.amazon.awssdk.services.dynamodb.model.ScanResponse;
import java.util.Map;
import java.util.Set;

/**
 * Before running this Java V2 code example, set up your development
 * environment, including your credentials.
 *
 * For more information, see the following documentation topic:
 *
 * https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-
started.html
 *
 * To scan items from an Amazon DynamoDB table using the AWS SDK for Java V2,
 * its better practice to use the
 * Enhanced Client, See the EnhancedScanRecords example.
 */

public class DynamoDBScanItems {
    public static void main(String[] args) {

        final String usage = """

                Usage:
                    <tableName>
```

```
                Where:
                    tableName - The Amazon DynamoDB table to get information from
 (for example, Music3).
                """;

        if (args.length != 1) {
            System.out.println(usage);
            System.exit(1);
        }

        String tableName = args[0];
        Region region = Region.US_EAST_1;
        DynamoDbClient ddb = DynamoDbClient.builder()
                .region(region)
                .build();

        scanItems(ddb, tableName);
        ddb.close();
    }

    public static void scanItems(DynamoDbClient ddb, String tableName) {
        try {
            ScanRequest scanRequest = ScanRequest.builder()
                    .tableName(tableName)
                    .build();

            ScanResponse response = ddb.scan(scanRequest);
            for (Map<String, AttributeValue> item : response.items()) {
                Set<String> keys = item.keySet();
                for (String key : keys) {
                    System.out.println("The key name is " + key + "\n");
                    System.out.println("The value is " + item.get(key).s());
                }
            }

        } catch (DynamoDbException e) {
            e.printStackTrace();
            System.exit(1);
        }
    }
}
```

- For API details, see Scan in *AWS SDK for Java 2.x API Reference*.

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

This example uses the document client to simplify working with items in DynamoDB. For API details see ScanCommand.

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
import { DynamoDBDocumentClient, ScanCommand } from "@aws-sdk/lib-dynamodb";

const client = new DynamoDBClient({});
const docClient = DynamoDBDocumentClient.from(client);

export const main = async () => {
  const command = new ScanCommand({
    ProjectionExpression: "#Name, Color, AvgLifeSpan",
    ExpressionAttributeNames: { "#Name": "Name" },
    TableName: "Birds",
  });

  const response = await docClient.send(command);
  for (const bird of response.Items) {
    console.log(`${bird.Name} - (${bird.Color}, ${bird.AvgLifeSpan})`);
  }
  return response;
};
```

- For API details, see Scan in *AWS SDK for JavaScript API Reference*.

SDK for JavaScript (v2)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
// Load the AWS SDK for Node.js.
var AWS = require("aws-sdk");
// Set the AWS Region.
AWS.config.update({ region: "REGION" });

// Create DynamoDB service object.
var ddb = new AWS.DynamoDB({ apiVersion: "2012-08-10" });

const params = {
  // Specify which items in the results are returned.
  FilterExpression: "Subtitle = :topic AND Season = :s AND Episode = :e",
  // Define the expression attribute value, which are substitutes for the values
 you want to compare.
  ExpressionAttributeValues: {
    ":topic": { S: "SubTitle2" },
    ":s": { N: 1 },
    ":e": { N: 2 },
  },
  // Set the projection expression, which are the attributes that you want.
  ProjectionExpression: "Season, Episode, Title, Subtitle",
  TableName: "EPISODES_TABLE",
};

ddb.scan(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data);
    data.Items.forEach(function (element, index, array) {
      console.log(
        "printing",
        element.Title.S + " (" + element.Subtitle.S + ")"
      );
    });
  }
});
```

- For more information, see AWS SDK for JavaScript Developer Guide.
- For API details, see Scan in *AWS SDK for JavaScript API Reference*.

Kotlin

SDK for Kotlin

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
suspend fun scanItems(tableNameVal: String) {
    val request =
        ScanRequest {
            tableName = tableNameVal
        }

    DynamoDbClient.fromEnvironment { region = "us-east-1" }.use { ddb ->
        val response = ddb.scan(request)
        response.items?.forEach { item ->
            item.keys.forEach { key ->
                println("The key name is $key\n")
                println("The value is ${item[key]}")
            }
        }
    }
}
```

- For API details, see Scan in *AWS SDK for Kotlin API reference*.

PHP

SDK for PHP

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
        $yearsKey = [
            'Key' => [
                'year' => [
                    'N' => [
                        'minRange' => 1990,
                        'maxRange' => 1999,
                    ],
                ],
            ],
        ];
        $filter = "year between 1990 and 1999";
        echo "\nHere's a list of all the movies released in the 90s:\n";
        $result = $service->scan($tableName, $yearsKey, $filter);
        foreach ($result['Items'] as $movie) {
            $movie = $marshal->unmarshalItem($movie);
            echo $movie['title'] . "\n";
        }

    public function scan(string $tableName, array $key, string $filters)
    {
        $query = [
            'ExpressionAttributeNames' => ['#year' => 'year'],
            'ExpressionAttributeValues' => [
                ":min" => ['N' => '1990'],
                ":max" => ['N' => '1999'],
            ],
            'FilterExpression' => "#year between :min and :max",
            'TableName' => $tableName,
        ];
        return $this->dynamoDbClient->scan($query);
    }
```

- For API details, see Scan in *AWS SDK for PHP API Reference*.

PowerShell

Tools for PowerShell V4

Example 1: Returns all items in the Music table.

```
Invoke-DDBScan -TableName 'Music' | ConvertFrom-DDBItem
```

Output:

```
Name                           Value
----                           -----
Genre                          Country
Artist                         No One You Know
Price                          1.94
CriticRating                   9
SongTitle                      Somewhere Down The Road
AlbumTitle                     Somewhat Famous
Genre                          Country
Artist                         No One You Know
Price                          1.98
CriticRating                   8.4
SongTitle                      My Dog Spot
AlbumTitle                     Hey Now
```

Example 2: Returns items in the Music table with a CriticRating greater than or equal to nine.

```
$scanFilter = @{
        CriticRating = [Amazon.DynamoDBv2.Model.Condition]@{
            AttributeValueList = @(@{N = '9'})
            ComparisonOperator = 'GE'
        }
    }
    Invoke-DDBScan -TableName 'Music' -ScanFilter $scanFilter | ConvertFrom-
DDBItem
```

Output:

```
Name                           Value
----                           -----
Genre                          Country
Artist                         No One You Know
Price                          1.94
CriticRating                   9
SongTitle                      Somewhere Down The Road
AlbumTitle                     Somewhat Famous
```

- For API details, see Scan in *AWS Tools for PowerShell Cmdlet Reference (V4)*.

Tools for PowerShell V5

Example 1: Returns all items in the Music table.

```
Invoke-DDBScan -TableName 'Music' | ConvertFrom-DDBItem
```

Output:

```
Name                           Value
----                           -----
Genre                          Country
Artist                         No One You Know
Price                          1.94
CriticRating                   9
SongTitle                      Somewhere Down The Road
AlbumTitle                     Somewhat Famous
Genre                          Country
Artist                         No One You Know
Price                          1.98
CriticRating                   8.4
SongTitle                      My Dog Spot
AlbumTitle                     Hey Now
```

Example 2: Returns items in the Music table with a CriticRating greater than or equal to nine.

```
$scanFilter = @{
        CriticRating = [Amazon.DynamoDBv2.Model.Condition]@{
            AttributeValueList = @(@{N = '9'})
            ComparisonOperator = 'GE'
        }
    }
    Invoke-DDBScan -TableName 'Music' -ScanFilter $scanFilter | ConvertFrom-
DDBItem
```

Output:

```
Name                           Value
----                           -----
Genre                          Country
Artist                         No One You Know
Price                          1.94
```

```
CriticRating                    9
SongTitle                       Somewhere Down The Road
AlbumTitle                      Somewhat Famous
```

- For API details, see Scan in *AWS Tools for PowerShell Cmdlet Reference (V5)*.

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
class Movies:
    """Encapsulates an Amazon DynamoDB table of movie data.

    Example data structure for a movie record in this table:
        {
            "year": 1999,
            "title": "For Love of the Game",
            "info": {
                "directors": ["Sam Raimi"],
                "release_date": "1999-09-15T00:00:00Z",
                "rating": 6.3,
                "plot": "A washed up pitcher flashes through his career.",
                "rank": 4987,
                "running_time_secs": 8220,
                "actors": [
                    "Kevin Costner",
                    "Kelly Preston",
                    "John C. Reilly"
                ]
            }
        }
    """

    def __init__(self, dyn_resource):
        """
        :param dyn_resource: A Boto3 DynamoDB resource.
```

```
        """
        self.dyn_resource = dyn_resource
        # The table variable is set during the scenario in the call to
        # 'exists' if the table exists. Otherwise, it is set by 'create_table'.
        self.table = None


    def scan_movies(self, year_range):
        """
        Scans for movies that were released in a range of years.
        Uses a projection expression to return a subset of data for each movie.

        :param year_range: The range of years to retrieve.
        :return: The list of movies released in the specified years.
        """
        movies = []
        scan_kwargs = {
            "FilterExpression": Key("year").between(
                year_range["first"], year_range["second"]
            ),
            "ProjectionExpression": "#yr, title, info.rating",
            "ExpressionAttributeNames": {"#yr": "year"},
        }
        try:
            done = False
            start_key = None
            while not done:
                if start_key:
                    scan_kwargs["ExclusiveStartKey"] = start_key
                response = self.table.scan(**scan_kwargs)
                movies.extend(response.get("Items", []))
                start_key = response.get("LastEvaluatedKey", None)
                done = start_key is None
        except ClientError as err:
            logger.error(
                "Couldn't scan for movies. Here's why: %s: %s",
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise

        return movies
```

- For API details, see Scan in *AWS SDK for Python (Boto3) API Reference*.

Ruby

SDK for Ruby

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
class DynamoDBBasics
  attr_reader :dynamo_resource, :table

  def initialize(table_name)
    client = Aws::DynamoDB::Client.new(region: 'us-east-1')
    @dynamo_resource = Aws::DynamoDB::Resource.new(client: client)
    @table = @dynamo_resource.table(table_name)
  end

  # Scans for movies that were released in a range of years.
  # Uses a projection expression to return a subset of data for each movie.
  #
  # @param year_range [Hash] The range of years to retrieve.
  # @return [Array] The list of movies released in the specified years.
  def scan_items(year_range)
    movies = []
    scan_hash = {
      filter_expression: '#yr between :start_yr and :end_yr',
      projection_expression: '#yr, title, info.rating',
      expression_attribute_names: { '#yr' => 'year' },
      expression_attribute_values: {
        ':start_yr' => year_range[:start], ':end_yr' => year_range[:end]
      }
    }
    done = false
    start_key = nil
    until done
      scan_hash[:exclusive_start_key] = start_key unless start_key.nil?
```

```
      response = @table.scan(scan_hash)
      movies.concat(response.items) unless response.items.empty?
      start_key = response.last_evaluated_key
      done = start_key.nil?
    end
  rescue Aws::DynamoDB::Errors::ServiceError => e
    puts("Couldn't scan for movies. Here's why:")
    puts("\t#{e.code}: #{e.message}")
    raise
  else
    movies
  end
```

- For API details, see Scan in *AWS SDK for Ruby API Reference*.

Rust

SDK for Rust

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
pub async fn list_items(client: &Client, table: &str, page_size: Option<i32>) ->
 Result<(), Error> {
    let page_size = page_size.unwrap_or(10);
    let items: Result<Vec<_>, _> = client
        .scan()
        .table_name(table)
        .limit(page_size)
        .into_paginator()
        .items()
        .send()
        .collect()
        .await;

    println!("Items in table (up to {page_size}):");
    for item in items? {
        println!("   {:?}", item);
```

```
    }

    Ok(())
}
```

- For API details, see Scan in *AWS SDK for Rust API reference*.

SAP ABAP

SDK for SAP ABAP

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    TRY.
        " Scan movies for rating greater than or equal to the rating specified
        DATA(lt_attributelist) = VALUE /aws1/
cl_dynattributevalue=>tt_attributevaluelist(
            ( NEW /aws1/cl_dynattributevalue( iv_n = |{ iv_rating }| ) ) ).
        DATA(lt_filter_conditions) = VALUE /aws1/
cl_dyncondition=>tt_filterconditionmap(
          ( VALUE /aws1/cl_dyncondition=>ts_filterconditionmap_maprow(
          key = 'rating'
          value = NEW /aws1/cl_dyncondition(
          it_attributevaluelist = lt_attributelist
          iv_comparisonoperator = |GE|
          ) ) ) ).
        oo_scan_result = lo_dyn->scan( iv_tablename = iv_table_name
          it_scanfilter = lt_filter_conditions ).
        DATA(lt_items) = oo_scan_result->get_items( ).
        LOOP AT lt_items INTO DATA(lo_item).
          " You can loop over to get individual attributes.
          DATA(lo_title) = lo_item[ key = 'title' ]-value.
          DATA(lo_year) = lo_item[ key = 'year' ]-value.
        ENDLOOP.
        DATA(lv_count) = oo_scan_result->get_count( ).
        MESSAGE 'Found ' && lv_count && ' items' TYPE 'I'.
      CATCH /aws1/cx_dynresourcenotfoundex.
```

```
        MESSAGE 'The table or index does not exist' TYPE 'E'.
    ENDTRY.
```

- For API details, see Scan in *AWS SDK for SAP ABAP API reference*.

Swift

SDK for Swift

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import AWSDynamoDB


    /// Return an array of `Movie` objects released in the specified range of
    /// years.
    ///
    /// - Parameters:
    ///   - firstYear: The first year of movies to return.
    ///   - lastYear: The last year of movies to return.
    ///   - startKey: A starting point to resume processing; always use `nil`.
    ///
    /// - Returns: An array of `Movie` objects describing the matching movies.
    ///
    /// > Note: The `startKey` parameter is used by this function when
    ///   recursively calling itself, and should always be `nil` when calling
    ///   directly.
    ///
    func getMovies(firstYear: Int, lastYear: Int,
                   startKey: [Swift.String: DynamoDBClientTypes.AttributeValue]?
 = nil)
        async throws -> [Movie]
    {
        do {
            var movieList: [Movie] = []

            guard let client = self.ddbClient else {
```

```
                throw MoviesError.UninitializedClient
            }

            let input = ScanInput(
                consistentRead: true,
                exclusiveStartKey: startKey,
                expressionAttributeNames: [
                    "#y": "year" // `year` is a reserved word, so use `#y`
instead.
                ],
                expressionAttributeValues: [
                    ":y1": .n(String(firstYear)),
                    ":y2": .n(String(lastYear))
                ],
                filterExpression: "#y BETWEEN :y1 AND :y2",
                tableName: self.tableName
            )

            let pages = client.scanPaginated(input: input)

            for try await page in pages {
                guard let items = page.items else {
                    print("Error: no items returned.")
                    continue
                }

                // Build an array of `Movie` objects for the returned items.

                for item in items {
                    let movie = try Movie(withItem: item)
                    movieList.append(movie)
                }
            }
            return movieList

        } catch {
            print("ERROR: getMovies with scan:", dump(error))
            throw error
        }
    }
```

- For API details, see Scan in *AWS SDK for Swift API reference*.

For a complete list of AWS SDK developer guides and code examples, see Using DynamoDB with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use `UpdateItem` with an AWS SDK or CLI

The following code examples show how to use `UpdateItem`.

Action examples are code excerpts from larger programs and must be run in context. You can see this action in context in the following code examples:

- Learn the basics
- Conditionally update an item's TTL
- Count expression operators
- Create and manage MRSC global tables
- Perform list operations
- Perform map operations
- Perform set operations
- Understand update expression order
- Update an item's TTL
- Use atomic counter operations
- Use conditional operations
- Use expression attribute names

.NET

SDK for .NET (v4)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
/// <summary>
```

```
    /// Updates an existing item in the movies table.
    /// </summary>
    /// <param name="newMovie">A Movie object containing information for
    /// the movie to update.</param>
    /// <param name="newInfo">A MovieInfo object that contains the
    /// information that will be changed.</param>
    /// <param name="tableName">The name of the table that contains the movie.</
param>
    /// <returns>A Boolean value that indicates the success of the operation.</
returns>
    public async Task<bool> UpdateItemAsync(
        Movie newMovie,
        MovieInfo newInfo,
        string tableName)
    {
        try
        {
            var key = new Dictionary<string, AttributeValue>
            {
                ["title"] = new AttributeValue { S = newMovie.Title },
                ["year"] = new AttributeValue { N = newMovie.Year.ToString() },
            };
            var updates = new Dictionary<string, AttributeValueUpdate>
            {
                ["info.plot"] = new AttributeValueUpdate
                {
                    Action = AttributeAction.PUT,
                    Value = new AttributeValue { S = newInfo.Plot },
                },

                ["info.rating"] = new AttributeValueUpdate
                {
                    Action = AttributeAction.PUT,
                    Value = new AttributeValue { N = newInfo.Rank.ToString() },
                },
            };

            var request = new UpdateItemRequest
            {
                AttributeUpdates = updates,
                Key = key,
                TableName = tableName,
            };
```

```
            await _amazonDynamoDB.UpdateItemAsync(request);
            return true;
        }
        catch (ResourceNotFoundException ex)
        {
            Console.WriteLine($"Table {tableName} or item was not found.
{ex.Message}");
            return false;
        }
        catch (AmazonDynamoDBException ex)
        {
            Console.WriteLine($"An Amazon DynamoDB error occurred while updating
item. {ex.Message}");
            throw;
        }
        catch (Exception ex)
        {
            Console.WriteLine($"An error occurred while updating item.
{ex.Message}");
            throw;
        }
    }
```

- For API details, see UpdateItem in *AWS SDK for .NET API Reference*.

Bash

AWS CLI with Bash script

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
#############################################################################
# function dynamodb_update_item
#
# This function updates an item in a DynamoDB table.
#
```

```
#
# Parameters:
#       -n table_name  -- The name of the table.
#       -k keys  -- Path to json file containing the keys that identify the item
 to update.
#       -e update expression  -- An expression that defines one or more
 attributes to be updated.
#       -v values  -- Path to json file containing the update values.
#
#  Returns:
#       0 - If successful.
#       1 - If it fails.
###############################################################################
function dynamodb_update_item() {
  local table_name keys update_expression values response
  local option OPTARG # Required to use getopts command in a function.

  ######################################
  # Function usage explanation
  ######################################
  function usage() {
    echo "function dynamodb_update_item"
    echo "Update an item in a DynamoDB table."
    echo " -n table_name  -- The name of the table."
    echo " -k keys  -- Path to json file containing the keys that identify the
 item to update."
    echo " -e update expression  -- An expression that defines one or more
 attributes to be updated."
    echo " -v values  -- Path to json file containing the update values."
    echo ""
  }

  while getopts "n:k:e:v:h" option; do
    case "${option}" in
      n) table_name="${OPTARG}" ;;
      k) keys="${OPTARG}" ;;
      e) update_expression="${OPTARG}" ;;
      v) values="${OPTARG}" ;;
      h)
        usage
        return 0
        ;;
      \?)
        echo "Invalid parameter"
```

```
      usage
      return 1
      ;;
  esac
done
export OPTIND=1

if [[ -z "$table_name" ]]; then
  errecho "ERROR: You must provide a table name with the -n parameter."
  usage
  return 1
fi

if [[ -z "$keys" ]]; then
  errecho "ERROR: You must provide a keys json file path the -k parameter."
  usage
  return 1
fi
if [[ -z "$update_expression" ]]; then
  errecho "ERROR: You must provide an update expression with the -e parameter."
  usage
  return 1
fi

if [[ -z "$values" ]]; then
  errecho "ERROR: You must provide a values json file path the -v parameter."
  usage
  return 1
fi

iecho "Parameters:\n"
iecho "    table_name:   $table_name"
iecho "    keys:   $keys"
iecho "    update_expression:   $update_expression"
iecho "    values:   $values"

response=$(aws dynamodb update-item \
  --table-name "$table_name" \
  --key file://"$keys" \
  --update-expression "$update_expression" \
  --expression-attribute-values file://"$values")

local error_code=${?}
```

```
  if [[ $error_code -ne 0 ]]; then
    aws_cli_error_log $error_code
    errecho "ERROR: AWS reports update-item operation failed.$response"
    return 1
  fi

  return 0

}
```

The utility functions used in this example.

```
###############################################################################
# function iecho
#
# This function enables the script to display the specified text only if
# the global variable $VERBOSE is set to true.
###############################################################################
function iecho() {
  if [[ $VERBOSE == true ]]; then
    echo "$@"
  fi
}

###############################################################################
# function errecho
#
# This function outputs everything sent to it to STDERR (standard error output).
###############################################################################
function errecho() {
  printf "%s\n" "$*" 1>&2
}

###############################################################################
# function aws_cli_error_log()
#
# This function is used to log the error messages from the AWS CLI.
#
# See https://docs.aws.amazon.com/cli/latest/topic/return-codes.html#cli-aws-
help-return-codes.
#
# The function expects the following argument:
```

```
#         $1 - The error code returned by the AWS CLI.
#
#  Returns:
#          0: - Success.
#
###############################################################################
function aws_cli_error_log() {
  local err_code=$1
  errecho "Error code : $err_code"
  if [ "$err_code" == 1 ]; then
    errecho "  One or more S3 transfers failed."
  elif [ "$err_code" == 2 ]; then
    errecho "  Command line failed to parse."
  elif [ "$err_code" == 130 ]; then
    errecho "  Process received SIGINT."
  elif [ "$err_code" == 252 ]; then
    errecho "  Command syntax invalid."
  elif [ "$err_code" == 253 ]; then
    errecho "  The system environment or configuration was invalid."
  elif [ "$err_code" == 254 ]; then
    errecho "  The service returned an error."
  elif [ "$err_code" == 255 ]; then
    errecho "  255 is a catch-all error."
  fi

  return 0
}
```

- For API details, see UpdateItem in *AWS CLI Command Reference*.

C++

SDK for C++

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
//! Update an Amazon DynamoDB table item.
```

```
/*!
  \sa updateItem()
  \param tableName: The table name.
  \param partitionKey: The partition key.
  \param partitionValue: The value for the partition key.
  \param attributeKey: The key for the attribute to be updated.
  \param attributeValue: The value for the attribute to be updated.
  \param clientConfiguration: AWS client configuration.
  \return bool: Function succeeded.
  */

/*
 *  The example code only sets/updates an attribute value. It processes
 *  the attribute value as a string, even if the value could be interpreted
 *  as a number. Also, the example code does not remove an existing attribute
 *  from the key value.
 */

bool AwsDoc::DynamoDB::updateItem(const Aws::String &tableName,
                                  const Aws::String &partitionKey,
                                  const Aws::String &partitionValue,
                                  const Aws::String &attributeKey,
                                  const Aws::String &attributeValue,
                                  const Aws::Client::ClientConfiguration
 &clientConfiguration) {
    Aws::DynamoDB::DynamoDBClient dynamoClient(clientConfiguration);

    // *** Define UpdateItem request arguments.
    // Define TableName argument.
    Aws::DynamoDB::Model::UpdateItemRequest request;
    request.SetTableName(tableName);

    // Define KeyName argument.
    Aws::DynamoDB::Model::AttributeValue attribValue;
    attribValue.SetS(partitionValue);
    request.AddKey(partitionKey, attribValue);

    // Construct the SET update expression argument.
    Aws::String update_expression("SET #a = :valueA");
    request.SetUpdateExpression(update_expression);

    // Construct attribute name argument.
    Aws::Map<Aws::String, Aws::String> expressionAttributeNames;
    expressionAttributeNames["#a"] = attributeKey;
```

```
    request.SetExpressionAttributeNames(expressionAttributeNames);

    // Construct attribute value argument.
    Aws::DynamoDB::Model::AttributeValue attributeUpdatedValue;
    attributeUpdatedValue.SetS(attributeValue);
    Aws::Map<Aws::String, Aws::DynamoDB::Model::AttributeValue>
 expressionAttributeValues;
    expressionAttributeValues[":valueA"] = attributeUpdatedValue;
    request.SetExpressionAttributeValues(expressionAttributeValues);

    // Update the item.
    const Aws::DynamoDB::Model::UpdateItemOutcome &outcome =
 dynamoClient.UpdateItem(
            request);
    if (outcome.IsSuccess()) {
        std::cout << "Item was updated" << std::endl;
    } else {
        std::cerr << outcome.GetError().GetMessage() << std::endl;
        return false;
    }

    return waitTableActive(tableName, dynamoClient);
}
```

Code that waits for the table to become active.

```
//! Query a newly created DynamoDB table until it is active.
/*!
  \sa waitTableActive()
  \param waitTableActive: The DynamoDB table's name.
  \param dynamoClient: A DynamoDB client.
  \return bool: Function succeeded.
*/
bool AwsDoc::DynamoDB::waitTableActive(const Aws::String &tableName,
                                       const Aws::DynamoDB::DynamoDBClient
 &dynamoClient) {

    // Repeatedly call DescribeTable until table is ACTIVE.
    const int MAX_QUERIES = 20;
    Aws::DynamoDB::Model::DescribeTableRequest request;
    request.SetTableName(tableName);
```

```
    int count = 0;
    while (count < MAX_QUERIES) {
        const Aws::DynamoDB::Model::DescribeTableOutcome &result =
 dynamoClient.DescribeTable(
                request);
        if (result.IsSuccess()) {
            Aws::DynamoDB::Model::TableStatus status =
 result.GetResult().GetTable().GetTableStatus();

            if (Aws::DynamoDB::Model::TableStatus::ACTIVE != status) {
                std::this_thread::sleep_for(std::chrono::seconds(1));
            }
            else {
                return true;
            }
        }
        else {
            std::cerr << "Error DynamoDB::waitTableActive "
                      << result.GetError().GetMessage() << std::endl;
            return false;
        }
        count++;
    }
    return false;
}
```

- For API details, see UpdateItem in *AWS SDK for C++ API Reference*.

CLI

AWS CLI

Example 1: To update an item in a table

The following `update-item` example updates an item in the `MusicCollection` table. It adds a new attribute (`Year`) and modifies the `AlbumTitle` attribute. All of the attributes in the item, as they appear after the update, are returned in the response.

```
aws dynamodb update-item \
    --table-name MusicCollection \
    --key file://key.json \
    --update-expression "SET #Y = :y, #AT = :t" \
```

```
    --expression-attribute-names file://expression-attribute-names.json \
    --expression-attribute-values file://expression-attribute-values.json  \
    --return-values ALL_NEW \
    --return-consumed-capacity TOTAL \
    --return-item-collection-metrics SIZE
```

Contents of `key.json`:

```
{
    "Artist": {"S": "Acme Band"},
    "SongTitle": {"S": "Happy Day"}
}
```

Contents of `expression-attribute-names.json`:

```
{
    "#Y":"Year", "#AT":"AlbumTitle"
}
```

Contents of `expression-attribute-values.json`:

```
{
    ":y":{"N": "2015"},
    ":t":{"S": "Louder Than Ever"}
}
```

Output:

```
{
    "Attributes": {
        "AlbumTitle": {
            "S": "Louder Than Ever"
        },
        "Awards": {
            "N": "10"
        },
        "Artist": {
            "S": "Acme Band"
        },
        "Year": {
            "N": "2015"
        },
```

```
        "SongTitle": {
            "S": "Happy Day"
        }
    },
    "ConsumedCapacity": {
        "TableName": "MusicCollection",
        "CapacityUnits": 3.0
    },
    "ItemCollectionMetrics": {
        "ItemCollectionKey": {
            "Artist": {
                "S": "Acme Band"
            }
        },
        "SizeEstimateRangeGB": [
            0.0,
            1.0
        ]
    }
}
```

For more information, see Writing an Item in the *Amazon DynamoDB Developer Guide*.

Example 2: To update an item conditionally

The following example updates an item in the `MusicCollection` table, but only if the existing item does not already have a `Year` attribute.

```
aws dynamodb update-item \
    --table-name MusicCollection \
    --key file://key.json \
    --update-expression "SET #Y = :y, #AT = :t" \
    --expression-attribute-names file://expression-attribute-names.json \
    --expression-attribute-values file://expression-attribute-values.json  \
    --condition-expression "attribute_not_exists(#Y)"
```

Contents of `key.json`:

```
{
    "Artist": {"S": "Acme Band"},
    "SongTitle": {"S": "Happy Day"}
}
```

Contents of `expression-attribute-names.json`:

```
{
    "#Y":"Year",
    "#AT":"AlbumTitle"
}
```

Contents of `expression-attribute-values.json`:

```
{
    ":y":{"N": "2015"},
    ":t":{"S": "Louder Than Ever"}
}
```

If the item already has a `Year` attribute, DynamoDB returns the following output.

```
An error occurred (ConditionalCheckFailedException) when calling the UpdateItem
 operation: The conditional request failed
```

For more information, see Writing an Item in the *Amazon DynamoDB Developer Guide*.

- For API details, see UpdateItem in *AWS CLI Command Reference*.

Go

SDK for Go V2

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import (
 "context"
 "errors"
 "log"
 "time"

 "github.com/aws/aws-sdk-go-v2/aws"
```

```
 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/attributevalue"
 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/expression"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb/types"
)

// TableBasics encapsulates the Amazon DynamoDB service actions used in the
 examples.
// It contains a DynamoDB service client that is used to act on the specified
 table.
type TableBasics struct {
 DynamoDbClient *dynamodb.Client
 TableName      string
}



// UpdateMovie updates the rating and plot of a movie that already exists in the
// DynamoDB table. This function uses the `expression` package to build the
 update
// expression.
func (basics TableBasics) UpdateMovie(ctx context.Context, movie Movie)
 (map[string]map[string]interface{}, error) {
 var err error
 var response *dynamodb.UpdateItemOutput
 var attributeMap map[string]map[string]interface{}
 update := expression.Set(expression.Name("info.rating"),
 expression.Value(movie.Info["rating"]))
 update.Set(expression.Name("info.plot"), expression.Value(movie.Info["plot"]))
 expr, err := expression.NewBuilder().WithUpdate(update).Build()
 if err != nil {
  log.Printf("Couldn't build expression for update. Here's why: %v\n", err)
 } else {
  response, err = basics.DynamoDbClient.UpdateItem(ctx,
 &dynamodb.UpdateItemInput{
   TableName:                 aws.String(basics.TableName),
   Key:                       movie.GetKey(),
   ExpressionAttributeNames:  expr.Names(),
   ExpressionAttributeValues: expr.Values(),
   UpdateExpression:          expr.Update(),
   ReturnValues:              types.ReturnValueUpdatedNew,
  })
  if err != nil {
   log.Printf("Couldn't update movie %v. Here's why: %v\n", movie.Title, err)
```

```
  } else {
   err = attributevalue.UnmarshalMap(response.Attributes, &attributeMap)
   if err != nil {
    log.Printf("Couldn't unmarshall update response. Here's why: %v\n", err)
   }
  }
 }
 return attributeMap, err
}
```

Define a Movie struct that is used in this example.

```
import (
 "archive/zip"
 "bytes"
 "encoding/json"
 "fmt"
 "io"
 "log"
 "net/http"

 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/attributevalue"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb/types"
)

// Movie encapsulates data about a movie. Title and Year are the composite
 primary key
// of the movie in Amazon DynamoDB. Title is the sort key, Year is the partition
 key,
// and Info is additional data.
type Movie struct {
 Title string                 `dynamodbav:"title"`
 Year  int                    `dynamodbav:"year"`
 Info  map[string]interface{} `dynamodbav:"info"`
}

// GetKey returns the composite primary key of the movie in a format that can be
// sent to DynamoDB.
func (movie Movie) GetKey() map[string]types.AttributeValue {
 title, err := attributevalue.Marshal(movie.Title)
```

```
 if err != nil {
  panic(err)
 }
 year, err := attributevalue.Marshal(movie.Year)
 if err != nil {
  panic(err)
 }
 return map[string]types.AttributeValue{"title": title, "year": year}
}

// String returns the title, year, rating, and plot of a movie, formatted for the
 example.
func (movie Movie) String() string {
 return fmt.Sprintf("%v\n\tReleased: %v\n\tRating: %v\n\tPlot: %v\n",
  movie.Title, movie.Year, movie.Info["rating"], movie.Info["plot"])
}
```

- For API details, see UpdateItem in *AWS SDK for Go API Reference*.

Java

SDK for Java 2.x

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Updates an item in a table using DynamoDbClient.

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.AttributeAction;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.AttributeValueUpdate;
import software.amazon.awssdk.services.dynamodb.model.UpdateItemRequest;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import java.util.HashMap;
```

```
/**
 * Before running this Java V2 code example, set up your development
 * environment, including your credentials.
 *
 * For more information, see the following documentation topic:
 *
 * https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-
started.html
 *
 * To update an Amazon DynamoDB table using the AWS SDK for Java V2, its better
 * practice to use the
 * Enhanced Client, See the EnhancedModifyItem example.
 */
public class UpdateItem {
    public static void main(String[] args) {
        final String usage = """

                Usage:
                    <tableName> <key> <keyVal> <name> <updateVal>

                Where:
                    tableName - The Amazon DynamoDB table (for example, Music3).
                    key - The name of the key in the table (for example, Artist).
                    keyVal - The value of the key (for example, Famous Band).
                    name - The name of the column where the value is updated (for
 example, Awards).
                    updateVal - The value used to update an item (for example,
 14).
                 Example:
                    UpdateItem Music3 Artist Famous Band Awards 14
                """;

        if (args.length != 5) {
            System.out.println(usage);
            System.exit(1);
        }

        String tableName = args[0];
        String key = args[1];
        String keyVal = args[2];
        String name = args[3];
        String updateVal = args[4];

        Region region = Region.US_EAST_1;
```

```
        DynamoDbClient ddb = DynamoDbClient.builder()
                .region(region)
                .build();
        updateTableItem(ddb, tableName, key, keyVal, name, updateVal);
        ddb.close();
    }

    public static void updateTableItem(DynamoDbClient ddb,
            String tableName,
            String key,
            String keyVal,
            String name,
            String updateVal) {

        HashMap<String, AttributeValue> itemKey = new HashMap<>();
        itemKey.put(key, AttributeValue.builder()
                .s(keyVal)
                .build());

        HashMap<String, AttributeValueUpdate> updatedValues = new HashMap<>();
        updatedValues.put(name, AttributeValueUpdate.builder()
                .value(AttributeValue.builder().s(updateVal).build())
                .action(AttributeAction.PUT)
                .build());

        UpdateItemRequest request = UpdateItemRequest.builder()
                .tableName(tableName)
                .key(itemKey)
                .attributeUpdates(updatedValues)
                .build();

        try {
            ddb.updateItem(request);
        } catch (DynamoDbException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
        System.out.println("The Amazon DynamoDB table was updated!");
    }
}
```

- For API details, see UpdateItem in *AWS SDK for Java 2.x API Reference*.

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

This example uses the document client to simplify working with items in DynamoDB. For API details see UpdateCommand.

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
import { DynamoDBDocumentClient, UpdateCommand } from "@aws-sdk/lib-dynamodb";

const client = new DynamoDBClient({});
const docClient = DynamoDBDocumentClient.from(client);

export const main = async () => {
  const command = new UpdateCommand({
    TableName: "Dogs",
    Key: {
      Breed: "Labrador",
    },
    UpdateExpression: "set Color = :color",
    ExpressionAttributeValues: {
      ":color": "black",
    },
    ReturnValues: "ALL_NEW",
  });

  const response = await docClient.send(command);
  console.log(response);
  return response;
};
```

- For API details, see UpdateItem in *AWS SDK for JavaScript API Reference*.

Kotlin

SDK for Kotlin

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
suspend fun updateTableItem(
    tableNameVal: String,
    keyName: String,
    keyVal: String,
    name: String,
    updateVal: String,
) {
    val itemKey = mutableMapOf<String, AttributeValue>()
    itemKey[keyName] = AttributeValue.S(keyVal)

    val updatedValues = mutableMapOf<String, AttributeValueUpdate>()
    updatedValues[name] =
        AttributeValueUpdate {
            value = AttributeValue.S(updateVal)
            action = AttributeAction.Put
        }

    val request =
        UpdateItemRequest {
            tableName = tableNameVal
            key = itemKey
            attributeUpdates = updatedValues
        }

    DynamoDbClient.fromEnvironment { region = "us-east-1" }.use { ddb ->
        ddb.updateItem(request)
        println("Item in $tableNameVal was updated")
    }
}
```

- For API details, see UpdateItem in *AWS SDK for Kotlin API reference*.

PHP

SDK for PHP

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
        echo "What rating would you like to give {$movie['Item']['title']['S']}?
\n";
        $rating = 0;
        while (!is_numeric($rating) || intval($rating) != $rating || $rating < 1
 || $rating > 10) {
            $rating = testable_readline("Rating (1-10): ");
        }
        $service->updateItemAttributeByKey($tableName, $key, 'rating', 'N',
 $rating);

    public function updateItemAttributeByKey(
        string $tableName,
        array $key,
        string $attributeName,
        string $attributeType,
        string $newValue
    ) {
        $this->dynamoDbClient->updateItem([
            'Key' => $key['Item'],
            'TableName' => $tableName,
            'UpdateExpression' => "set #NV=:NV",
            'ExpressionAttributeNames' => [
                '#NV' => $attributeName,
            ],
            'ExpressionAttributeValues' => [
                ':NV' => [
                    $attributeType => $newValue
                ]
            ],
        ]);
    }
```

- For API details, see UpdateItem in *AWS SDK for PHP API Reference*.

PowerShell

Tools for PowerShell V4

Example 1: Sets the genre attribute to 'Rap' on the DynamoDB item with the partition key SongTitle and the sort key Artist.

```
$key = @{
    SongTitle = 'Somewhere Down The Road'
    Artist = 'No One You Know'
} | ConvertTo-DDBItem

$updateDdbItem = @{
    TableName = 'Music'
    Key = $key
    UpdateExpression = 'set Genre = :val1'
    ExpressionAttributeValue = (@{
        ':val1' = ([Amazon.DynamoDBv2.Model.AttributeValue]'Rap')
    })
}
Update-DDBItem @updateDdbItem
```

Output:

```
Name                           Value
----                           -----
Genre                          Rap
```

- For API details, see UpdateItem in *AWS Tools for PowerShell Cmdlet Reference (V4)*.

Tools for PowerShell V5

Example 1: Sets the genre attribute to 'Rap' on the DynamoDB item with the partition key SongTitle and the sort key Artist.

```
$key = @{
    SongTitle = 'Somewhere Down The Road'
    Artist = 'No One You Know'
} | ConvertTo-DDBItem
```

```
$updateDdbItem = @{
    TableName = 'Music'
    Key = $key
    UpdateExpression = 'set Genre = :val1'
    ExpressionAttributeValue = (@{
        ':val1' = ([Amazon.DynamoDBv2.Model.AttributeValue]'Rap')
    })
}
Update-DDBItem @updateDdbItem
```

Output:

```
Name                           Value
----                           -----
Genre                          Rap
```

- For API details, see UpdateItem in *AWS Tools for PowerShell Cmdlet Reference (V5)*.

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Update an item by using an update expression.

```
class Movies:
    """Encapsulates an Amazon DynamoDB table of movie data.

    Example data structure for a movie record in this table:
        {
            "year": 1999,
            "title": "For Love of the Game",
            "info": {
                "directors": ["Sam Raimi"],
                "release_date": "1999-09-15T00:00:00Z",
                "rating": 6.3,
```

```
                "plot": "A washed up pitcher flashes through his career.",
                "rank": 4987,
                "running_time_secs": 8220,
                "actors": [
                    "Kevin Costner",
                    "Kelly Preston",
                    "John C. Reilly"
                ]
            }
        }
    """

    def __init__(self, dyn_resource):
        """
        :param dyn_resource: A Boto3 DynamoDB resource.
        """
        self.dyn_resource = dyn_resource
        # The table variable is set during the scenario in the call to
        # 'exists' if the table exists. Otherwise, it is set by 'create_table'.
        self.table = None


    def update_movie(self, title, year, rating, plot):
        """
        Updates rating and plot data for a movie in the table.

        :param title: The title of the movie to update.
        :param year: The release year of the movie to update.
        :param rating: The updated rating to the give the movie.
        :param plot: The updated plot summary to give the movie.
        :return: The fields that were updated, with their new values.
        """
        try:
            response = self.table.update_item(
                Key={"year": year, "title": title},
                UpdateExpression="set info.rating=:r, info.plot=:p",
                ExpressionAttributeValues={":r": Decimal(str(rating)), ":p":
plot},
                ReturnValues="UPDATED_NEW",
            )
        except ClientError as err:
            logger.error(
                "Couldn't update movie %s in table %s. Here's why: %s: %s",
                title,
```

```
                self.table.name,
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
        else:
            return response["Attributes"]
```

Update an item by using an update expression that includes an arithmetic operation.

```
class UpdateQueryWrapper:
    def __init__(self, table):
        self.table = table


    def update_rating(self, title, year, rating_change):
        """
        Updates the quality rating of a movie in the table by using an arithmetic
        operation in the update expression. By specifying an arithmetic
 operation,
        you can adjust a value in a single request, rather than first getting its
        value and then setting its new value.

        :param title: The title of the movie to update.
        :param year: The release year of the movie to update.
        :param rating_change: The amount to add to the current rating for the
 movie.
        :return: The updated rating.
        """
        try:
            response = self.table.update_item(
                Key={"year": year, "title": title},
                UpdateExpression="set info.rating = info.rating + :val",
                ExpressionAttributeValues={":val": Decimal(str(rating_change))},
                ReturnValues="UPDATED_NEW",
            )
        except ClientError as err:
            logger.error(
                "Couldn't update movie %s in table %s. Here's why: %s: %s",
                title,
                self.table.name,
```

```
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
        else:
            return response["Attributes"]
```

Update an item only when it meets certain conditions.

```
class UpdateQueryWrapper:
    def __init__(self, table):
        self.table = table


    def remove_actors(self, title, year, actor_threshold):
        """
        Removes an actor from a movie, but only when the number of actors is
 greater
        than a specified threshold. If the movie does not list more than the
 threshold,
        no actors are removed.

        :param title: The title of the movie to update.
        :param year: The release year of the movie to update.
        :param actor_threshold: The threshold of actors to check.
        :return: The movie data after the update.
        """
        try:
            response = self.table.update_item(
                Key={"year": year, "title": title},
                UpdateExpression="remove info.actors[0]",
                ConditionExpression="size(info.actors) > :num",
                ExpressionAttributeValues={":num": actor_threshold},
                ReturnValues="ALL_NEW",
            )
        except ClientError as err:
            if err.response["Error"]["Code"] ==
 "ConditionalCheckFailedException":
                logger.warning(
                    "Didn't update %s because it has fewer than %s actors.",
                    title,
```

```
                    actor_threshold + 1,
                )
            else:
                logger.error(
                    "Couldn't update movie %s. Here's why: %s: %s",
                    title,
                    err.response["Error"]["Code"],
                    err.response["Error"]["Message"],
                )
                raise
        else:
            return response["Attributes"]
```

- For API details, see UpdateItem in *AWS SDK for Python (Boto3) API Reference*.

Ruby

SDK for Ruby

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
class DynamoDBBasics
  attr_reader :dynamo_resource, :table

  def initialize(table_name)
    client = Aws::DynamoDB::Client.new(region: 'us-east-1')
    @dynamo_resource = Aws::DynamoDB::Resource.new(client: client)
    @table = @dynamo_resource.table(table_name)
  end

  # Updates rating and plot data for a movie in the table.
  #
  # @param movie [Hash] The title, year, plot, rating of the movie.
  def update_item(movie)
    response = @table.update_item(
      key: { 'year' => movie[:year], 'title' => movie[:title] },
```

```
      update_expression: 'set info.rating=:r',
      expression_attribute_values: { ':r' => movie[:rating] },
      return_values: 'UPDATED_NEW'
    )
  rescue Aws::DynamoDB::Errors::ServiceError => e
    puts("Couldn't update movie #{movie[:title]} (#{movie[:year]}) in table
#{@table.name}\n")
    puts("\t#{e.code}: #{e.message}")
    raise
  else
    response.attributes
  end
```

- For API details, see UpdateItem in *AWS SDK for Ruby API Reference*.

SAP ABAP

SDK for SAP ABAP

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    TRY.
        oo_output = lo_dyn->updateitem(
          iv_tablename        = iv_table_name
          it_key              = it_item_key
          it_attributeupdates = it_attribute_updates ).
        MESSAGE '1 item updated in DynamoDB Table' && iv_table_name TYPE 'I'.
      CATCH /aws1/cx_dyncondalcheckfaile00.
        MESSAGE 'A condition specified in the operation could not be evaluated.'
TYPE 'E'.
      CATCH /aws1/cx_dynresourcenotfoundex.
        MESSAGE 'The table or index does not exist' TYPE 'E'.
      CATCH /aws1/cx_dyntransactconflictex.
        MESSAGE 'Another transaction is using the item' TYPE 'E'.
    ENDTRY.
```

- For API details, see UpdateItem in *AWS SDK for SAP ABAP API reference*.

Swift

SDK for Swift

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import AWSDynamoDB


    /// Update the specified movie with new `rating` and `plot` information.
    ///
    /// - Parameters:
    ///   - title: The title of the movie to update.
    ///   - year: The release year of the movie to update.
    ///   - rating: The new rating for the movie.
    ///   - plot: The new plot summary string for the movie.
    ///
    /// - Returns: An array of mappings of attribute names to their new
    ///   listing each item actually changed. Items that didn't need to change
    ///   aren't included in this list. `nil` if no changes were made.
    ///
    func update(title: String, year: Int, rating: Double? = nil, plot: String? =
 nil) async throws
        -> [Swift.String: DynamoDBClientTypes.AttributeValue]?
    {
        do {
            guard let client = self.ddbClient else {
                throw MoviesError.UninitializedClient
            }

            // Build the update expression and the list of expression attribute
            // values. Include only the information that's changed.

            var expressionParts: [String] = []
            var attrValues: [Swift.String: DynamoDBClientTypes.AttributeValue] =
 [:]
```

```
            if rating != nil {
                expressionParts.append("info.rating=:r")
                attrValues[":r"] = .n(String(rating!))
            }
            if plot != nil {
                expressionParts.append("info.plot=:p")
                attrValues[":p"] = .s(plot!)
            }
            let expression = "set \(expressionParts.joined(separator: ", "))"

            let input = UpdateItemInput(
                // Create substitution tokens for the attribute values, to ensure
                // no conflicts in expression syntax.
                expressionAttributeValues: attrValues,
                // The key identifying the movie to update consists of the
release
                // year and title.
                key: [
                    "year": .n(String(year)),
                    "title": .s(title)
                ],
                returnValues: .updatedNew,
                tableName: self.tableName,
                updateExpression: expression
            )
            let output = try await client.updateItem(input: input)

            guard let attributes: [Swift.String:
DynamoDBClientTypes.AttributeValue] = output.attributes else {
                throw MoviesError.InvalidAttributes
            }
            return attributes
        } catch {
            print("ERROR: update:", dump(error))
            throw error
        }
    }
```

- For API details, see UpdateItem in *AWS SDK for Swift API reference*.

For a complete list of AWS SDK developer guides and code examples, see Using DynamoDB with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use `UpdateTable` with an AWS SDK or CLI

The following code examples show how to use `UpdateTable`.

Action examples are code excerpts from larger programs and must be run in context. You can see this action in context in the following code examples:

- Create and manage global tables demonstrating MREC
- Create and manage MRSC global tables
- Manage Global Secondary Indexes
- Update a table's warm throughput setting
- Work with global tables and multi-Region replication eventual consistency (MREC)
- Work with table encryption

C++

SDK for C++

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
//! Update a DynamoDB table.
/*!
  \sa updateTable()
  \param tableName: Name for the DynamoDB table.
  \param readCapacity: Provisioned read capacity.
  \param writeCapacity: Provisioned write capacity.
  \param clientConfiguration: AWS client configuration.
  \return bool: Function succeeded.
 */
bool AwsDoc::DynamoDB::updateTable(const Aws::String &tableName,
```

```
                                    long long readCapacity, long long
writeCapacity,
                                    const Aws::Client::ClientConfiguration
&clientConfiguration) {
   Aws::DynamoDB::DynamoDBClient dynamoClient(clientConfiguration);

   std::cout << "Updating " << tableName << " with new provisioned throughput
values"
             << std::endl;
   std::cout << "Read capacity : " << readCapacity << std::endl;
   std::cout << "Write capacity: " << writeCapacity << std::endl;

   Aws::DynamoDB::Model::UpdateTableRequest request;
   Aws::DynamoDB::Model::ProvisionedThroughput provisionedThroughput;

provisionedThroughput.WithReadCapacityUnits(readCapacity).WithWriteCapacityUnits(
           writeCapacity);

request.WithProvisionedThroughput(provisionedThroughput).WithTableName(tableName);

   const Aws::DynamoDB::Model::UpdateTableOutcome &outcome =
dynamoClient.UpdateTable(
           request);
   if (outcome.IsSuccess()) {
       std::cout << "Successfully updated the table." << std::endl;
   } else {
       const Aws::DynamoDB::DynamoDBError &error = outcome.GetError();
       if (error.GetErrorType() == Aws::DynamoDB::DynamoDBErrors::VALIDATION &&
           error.GetMessage().find("The provisioned throughput for the table
will not change") != std::string::npos) {
           std::cout << "The provisioned throughput for the table will not
change." << std::endl;
       } else {
           std::cerr << outcome.GetError().GetMessage() << std::endl;
           return false;
       }
   }

   return waitTableActive(tableName, dynamoClient);
}
```

Code that waits for the table to become active.

```
//! Query a newly created DynamoDB table until it is active.
/*!
  \sa waitTableActive()
  \param waitTableActive: The DynamoDB table's name.
  \param dynamoClient: A DynamoDB client.
  \return bool: Function succeeded.
*/
bool AwsDoc::DynamoDB::waitTableActive(const Aws::String &tableName,
                                       const Aws::DynamoDB::DynamoDBClient
 &dynamoClient) {

    // Repeatedly call DescribeTable until table is ACTIVE.
    const int MAX_QUERIES = 20;
    Aws::DynamoDB::Model::DescribeTableRequest request;
    request.SetTableName(tableName);

    int count = 0;
    while (count < MAX_QUERIES) {
        const Aws::DynamoDB::Model::DescribeTableOutcome &result =
 dynamoClient.DescribeTable(
                request);
        if (result.IsSuccess()) {
            Aws::DynamoDB::Model::TableStatus status =
 result.GetResult().GetTable().GetTableStatus();

            if (Aws::DynamoDB::Model::TableStatus::ACTIVE != status) {
                std::this_thread::sleep_for(std::chrono::seconds(1));
            }
            else {
                return true;
            }
        }
        else {
            std::cerr << "Error DynamoDB::waitTableActive "
                      << result.GetError().GetMessage() << std::endl;
            return false;
        }
        count++;
    }
    return false;
}
```

- For API details, see UpdateTable in *AWS SDK for C++ API Reference*.

CLI

AWS CLI

Example 1: To modify a table's billing mode

The following `update-table` example increases the provisioned read and write capacity on the `MusicCollection` table.

```
aws dynamodb update-table \
    --table-name MusicCollection \
    --billing-mode PROVISIONED \
    --provisioned-throughput ReadCapacityUnits=15,WriteCapacityUnits=10
```

Output:

```
{
    "TableDescription": {
        "AttributeDefinitions": [
            {
                "AttributeName": "AlbumTitle",
                "AttributeType": "S"
            },
            {
                "AttributeName": "Artist",
                "AttributeType": "S"
            },
            {
                "AttributeName": "SongTitle",
                "AttributeType": "S"
            }
        ],
        "TableName": "MusicCollection",
        "KeySchema": [
            {
                "AttributeName": "Artist",
                "KeyType": "HASH"
            },
            {
                "AttributeName": "SongTitle",
```

```
                "KeyType": "RANGE"
            }
        ],
        "TableStatus": "UPDATING",
        "CreationDateTime": "2020-05-26T15:59:49.473000-07:00",
        "ProvisionedThroughput": {
            "LastIncreaseDateTime": "2020-07-28T13:18:18.921000-07:00",
            "NumberOfDecreasesToday": 0,
            "ReadCapacityUnits": 15,
            "WriteCapacityUnits": 10
        },
        "TableSizeBytes": 182,
        "ItemCount": 2,
        "TableArn": "arn:aws:dynamodb:us-west-2:123456789012:table/
MusicCollection",
        "TableId": "abcd0123-01ab-23cd-0123-abcdef123456",
        "BillingModeSummary": {
            "BillingMode": "PROVISIONED",
            "LastUpdateToPayPerRequestDateTime":
 "2020-07-28T13:14:48.366000-07:00"
        }
    }
}
```

For more information, see Updating a Table in the *Amazon DynamoDB Developer Guide*.

Example 2: To create a global secondary index

The following example adds a global secondary index to the `MusicCollection` table.

```
aws dynamodb update-table \
    --table-name MusicCollection \
    --attribute-definitions AttributeName=AlbumTitle,AttributeType=S \
    --global-secondary-index-updates file://gsi-updates.json
```

Contents of `gsi-updates.json`:

```
[
    {
        "Create": {
            "IndexName": "AlbumTitle-index",
            "KeySchema": [
                {
```

```
                    "AttributeName": "AlbumTitle",
                    "KeyType": "HASH"
                }
            ],
            "ProvisionedThroughput": {
                "ReadCapacityUnits": 10,
                "WriteCapacityUnits": 10
            },
            "Projection": {
                "ProjectionType": "ALL"
            }
        }
    }
]
```

Output:

```
{
    "TableDescription": {
        "AttributeDefinitions": [
            {
                "AttributeName": "AlbumTitle",
                "AttributeType": "S"
            },
            {
                "AttributeName": "Artist",
                "AttributeType": "S"
            },
            {
                "AttributeName": "SongTitle",
                "AttributeType": "S"
            }
        ],
        "TableName": "MusicCollection",
        "KeySchema": [
            {
                "AttributeName": "Artist",
                "KeyType": "HASH"
            },
            {
                "AttributeName": "SongTitle",
                "KeyType": "RANGE"
            }
```

```
        ],
        "TableStatus": "UPDATING",
        "CreationDateTime": "2020-05-26T15:59:49.473000-07:00",
        "ProvisionedThroughput": {
            "LastIncreaseDateTime": "2020-07-28T12:59:17.537000-07:00",
            "NumberOfDecreasesToday": 0,
            "ReadCapacityUnits": 15,
            "WriteCapacityUnits": 10
        },
        "TableSizeBytes": 182,
        "ItemCount": 2,
        "TableArn": "arn:aws:dynamodb:us-west-2:123456789012:table/
MusicCollection",
        "TableId": "abcd0123-01ab-23cd-0123-abcdef123456",
        "BillingModeSummary": {
            "BillingMode": "PROVISIONED",
            "LastUpdateToPayPerRequestDateTime":
 "2020-07-28T13:14:48.366000-07:00"
        },
        "GlobalSecondaryIndexes": [
            {
                "IndexName": "AlbumTitle-index",
                "KeySchema": [
                    {
                        "AttributeName": "AlbumTitle",
                        "KeyType": "HASH"
                    }
                ],
                "Projection": {
                    "ProjectionType": "ALL"
                },
                "IndexStatus": "CREATING",
                "Backfilling": false,
                "ProvisionedThroughput": {
                    "NumberOfDecreasesToday": 0,
                    "ReadCapacityUnits": 10,
                    "WriteCapacityUnits": 10
                },
                "IndexSizeBytes": 0,
                "ItemCount": 0,
                "IndexArn": "arn:aws:dynamodb:us-west-2:123456789012:table/
MusicCollection/index/AlbumTitle-index"
            }
        ]
```

```
    }
}
```

For more information, see Updating a Table in the *Amazon DynamoDB Developer Guide*.

Example 3: To enable DynamoDB Streams on a table

The following command enables DynamoDB Streams on the `MusicCollection` table.

```
aws dynamodb update-table \
    --table-name MusicCollection \
    --stream-specification StreamEnabled=true,StreamViewType=NEW_IMAGE
```

Output:

```
{
    "TableDescription": {
        "AttributeDefinitions": [
            {
                "AttributeName": "AlbumTitle",
                "AttributeType": "S"
            },
            {
                "AttributeName": "Artist",
                "AttributeType": "S"
            },
            {
                "AttributeName": "SongTitle",
                "AttributeType": "S"
            }
        ],
        "TableName": "MusicCollection",
        "KeySchema": [
            {
                "AttributeName": "Artist",
                "KeyType": "HASH"
            },
            {
                "AttributeName": "SongTitle",
                "KeyType": "RANGE"
            }
        ],
        "TableStatus": "UPDATING",
```

```
        "CreationDateTime": "2020-05-26T15:59:49.473000-07:00",
        "ProvisionedThroughput": {
            "LastIncreaseDateTime": "2020-07-28T12:59:17.537000-07:00",
            "NumberOfDecreasesToday": 0,
            "ReadCapacityUnits": 15,
            "WriteCapacityUnits": 10
        },
        "TableSizeBytes": 182,
        "ItemCount": 2,
        "TableArn": "arn:aws:dynamodb:us-west-2:123456789012:table/
MusicCollection",
        "TableId": "abcd0123-01ab-23cd-0123-abcdef123456",
        "BillingModeSummary": {
            "BillingMode": "PROVISIONED",
            "LastUpdateToPayPerRequestDateTime":
 "2020-07-28T13:14:48.366000-07:00"
        },
        "LocalSecondaryIndexes": [
            {
                "IndexName": "AlbumTitleIndex",
                "KeySchema": [
                    {
                        "AttributeName": "Artist",
                        "KeyType": "HASH"
                    },
                    {
                        "AttributeName": "AlbumTitle",
                        "KeyType": "RANGE"
                    }
                ],
                "Projection": {
                    "ProjectionType": "INCLUDE",
                    "NonKeyAttributes": [
                        "Year",
                        "Genre"
                    ]
                },
                "IndexSizeBytes": 139,
                "ItemCount": 2,
                "IndexArn": "arn:aws:dynamodb:us-west-2:123456789012:table/
MusicCollection/index/AlbumTitleIndex"
            }
        ],
        "GlobalSecondaryIndexes": [
```

```
            {
                "IndexName": "AlbumTitle-index",
                "KeySchema": [
                    {
                        "AttributeName": "AlbumTitle",
                        "KeyType": "HASH"
                    }
                ],
                "Projection": {
                    "ProjectionType": "ALL"
                },
                "IndexStatus": "ACTIVE",
                "ProvisionedThroughput": {
                    "NumberOfDecreasesToday": 0,
                    "ReadCapacityUnits": 10,
                    "WriteCapacityUnits": 10
                },
                "IndexSizeBytes": 0,
                "ItemCount": 0,
                "IndexArn": "arn:aws:dynamodb:us-west-2:123456789012:table/
MusicCollection/index/AlbumTitle-index"
            }
        ],
        "StreamSpecification": {
            "StreamEnabled": true,
            "StreamViewType": "NEW_IMAGE"
        },
        "LatestStreamLabel": "2020-07-28T21:53:39.112",
        "LatestStreamArn": "arn:aws:dynamodb:us-west-2:123456789012:table/
MusicCollection/stream/2020-07-28T21:53:39.112"
    }
}
```

For more information, see Updating a Table in the *Amazon DynamoDB Developer Guide*.

Example 4: To enable server-side encryption

The following example enables server-side encryption on the `MusicCollection` table.

```
aws dynamodb update-table \
    --table-name MusicCollection \
    --sse-specification Enabled=true,SSEType=KMS
```

Output:


```
{
    "TableDescription": {
        "AttributeDefinitions": [
            {
                "AttributeName": "AlbumTitle",
                "AttributeType": "S"
            },
            {
                "AttributeName": "Artist",
                "AttributeType": "S"
            },
            {
                "AttributeName": "SongTitle",
                "AttributeType": "S"
            }
        ],
        "TableName": "MusicCollection",
        "KeySchema": [
            {
                "AttributeName": "Artist",
                "KeyType": "HASH"
            },
            {
                "AttributeName": "SongTitle",
                "KeyType": "RANGE"
            }
        ],
        "TableStatus": "ACTIVE",
        "CreationDateTime": "2020-05-26T15:59:49.473000-07:00",
        "ProvisionedThroughput": {
            "LastIncreaseDateTime": "2020-07-28T12:59:17.537000-07:00",
            "NumberOfDecreasesToday": 0,
            "ReadCapacityUnits": 15,
            "WriteCapacityUnits": 10
        },
        "TableSizeBytes": 182,
        "ItemCount": 2,
        "TableArn": "arn:aws:dynamodb:us-west-2:123456789012:table/
MusicCollection",
        "TableId": "abcd0123-01ab-23cd-0123-abcdef123456",
        "BillingModeSummary": {
            "BillingMode": "PROVISIONED",
```

```
            "LastUpdateToPayPerRequestDateTime":
 "2020-07-28T13:14:48.366000-07:00"
        },
        "LocalSecondaryIndexes": [
            {
                "IndexName": "AlbumTitleIndex",
                "KeySchema": [
                    {
                        "AttributeName": "Artist",
                        "KeyType": "HASH"
                    },
                    {
                        "AttributeName": "AlbumTitle",
                        "KeyType": "RANGE"
                    }
                ],
                "Projection": {
                    "ProjectionType": "INCLUDE",
                    "NonKeyAttributes": [
                        "Year",
                        "Genre"
                    ]
                },
                "IndexSizeBytes": 139,
                "ItemCount": 2,
                "IndexArn": "arn:aws:dynamodb:us-west-2:123456789012:table/
MusicCollection/index/AlbumTitleIndex"
            }
        ],
        "GlobalSecondaryIndexes": [
            {
                "IndexName": "AlbumTitle-index",
                "KeySchema": [
                    {
                        "AttributeName": "AlbumTitle",
                        "KeyType": "HASH"
                    }
                ],
                "Projection": {
                    "ProjectionType": "ALL"
                },
                "IndexStatus": "ACTIVE",
                "ProvisionedThroughput": {
                    "NumberOfDecreasesToday": 0,
```

```
                    "ReadCapacityUnits": 10,
                    "WriteCapacityUnits": 10
                },
                "IndexSizeBytes": 0,
                "ItemCount": 0,
                "IndexArn": "arn:aws:dynamodb:us-west-2:123456789012:table/
MusicCollection/index/AlbumTitle-index"
            }
        ],
        "StreamSpecification": {
            "StreamEnabled": true,
            "StreamViewType": "NEW_IMAGE"
        },
        "LatestStreamLabel": "2020-07-28T21:53:39.112",
        "LatestStreamArn": "arn:aws:dynamodb:us-west-2:123456789012:table/
MusicCollection/stream/2020-07-28T21:53:39.112",
        "SSEDescription": {
            "Status": "UPDATING"
        }
    }
}
```

For more information, see Updating a Table in the *Amazon DynamoDB Developer Guide*.

- For API details, see UpdateTable in *AWS CLI Command Reference*.

PowerShell

Tools for PowerShell V4

Example 1: Updates the provisioned throughput for the given table.

```
Update-DDBTable -TableName "myTable" -ReadCapacity 10 -WriteCapacity 5
```

- For API details, see UpdateTable in *AWS Tools for PowerShell Cmdlet Reference (V4)*.

Tools for PowerShell V5

Example 1: Updates the provisioned throughput for the given table.

```
Update-DDBTable -TableName "myTable" -ReadCapacity 10 -WriteCapacity 5
```

- For API details, see UpdateTable in *AWS Tools for PowerShell Cmdlet Reference (V5)*.

For a complete list of AWS SDK developer guides and code examples, see Using DynamoDB with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use `UpdateTimeToLive` with an AWS SDK or CLI

The following code examples show how to use `UpdateTimeToLive`.

Action examples are code excerpts from larger programs and must be run in context. You can see this action in context in the following code example:

- Work with Streams and Time-to-Live

CLI

AWS CLI

To update Time to Live settings on a table

The following `update-time-to-live` example enables Time to Live on the specified table.

```
aws dynamodb update-time-to-live \
    --table-name MusicCollection \
    --time-to-live-specification Enabled=true,AttributeName=ttl
```

Output:

```
{
    "TimeToLiveSpecification": {
        "Enabled": true,
        "AttributeName": "ttl"
    }
}
```

For more information, see Time to Live in the *Amazon DynamoDB Developer Guide*.

- For API details, see UpdateTimeToLive in *AWS CLI Command Reference*.

Java

SDK for Java 2.x

Enable TTL on an existing DynamoDB table using AWS SDK for Java 2.x.

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.ResourceNotFoundException;
import software.amazon.awssdk.services.dynamodb.model.TimeToLiveSpecification;
import software.amazon.awssdk.services.dynamodb.model.UpdateTimeToLiveRequest;
import software.amazon.awssdk.services.dynamodb.model.UpdateTimeToLiveResponse;

import java.util.logging.Level;
import java.util.logging.Logger;

    public UpdateTimeToLiveResponse enableTTL(final String tableName, final
 String attributeName, final Region region) {
        final TimeToLiveSpecification ttlSpec = TimeToLiveSpecification.builder()
            .attributeName(attributeName)
            .enabled(true)
            .build();

        final UpdateTimeToLiveRequest request = UpdateTimeToLiveRequest.builder()
            .tableName(tableName)
            .timeToLiveSpecification(ttlSpec)
            .build();

        try (DynamoDbClient ddb = dynamoDbClient != null
            ? dynamoDbClient
            : DynamoDbClient.builder().region(region).build()) {
            return ddb.updateTimeToLive(request);
        } catch (ResourceNotFoundException e) {
            System.err.format(TABLE_NOT_FOUND_ERROR, tableName);
            throw e;
        } catch (DynamoDbException e) {
            System.err.println(e.getMessage());
            throw e;
        }
    }
```

Disable TTL on an existing DynamoDB table using AWS SDK for Java 2.x.

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.ResourceNotFoundException;
import software.amazon.awssdk.services.dynamodb.model.TimeToLiveSpecification;
import software.amazon.awssdk.services.dynamodb.model.UpdateTimeToLiveRequest;
import software.amazon.awssdk.services.dynamodb.model.UpdateTimeToLiveResponse;

import java.util.logging.Level;
import java.util.logging.Logger;

    public UpdateTimeToLiveResponse disableTTL(
        final String tableName, final String attributeName, final Region region)
 {
        final TimeToLiveSpecification ttlSpec = TimeToLiveSpecification.builder()
            .attributeName(attributeName)
            .enabled(false)
            .build();

        final UpdateTimeToLiveRequest request = UpdateTimeToLiveRequest.builder()
            .tableName(tableName)
            .timeToLiveSpecification(ttlSpec)
            .build();

        try (DynamoDbClient ddb = dynamoDbClient != null
            ? dynamoDbClient
            : DynamoDbClient.builder().region(region).build()) {
            return ddb.updateTimeToLive(request);
        } catch (ResourceNotFoundException e) {
            System.err.format(TABLE_NOT_FOUND_ERROR, tableName);
            throw e;
        } catch (DynamoDbException e) {
            System.err.println(e.getMessage());
            throw e;
        }
    }
```

- For API details, see UpdateTimeToLive in *AWS SDK for Java 2.x API Reference*.

JavaScript

SDK for JavaScript (v3)

Enable TTL on an existing DynamoDB table.

```
import { DynamoDBClient, UpdateTimeToLiveCommand } from "@aws-sdk/client-
dynamodb";

export const enableTTL = async (tableName, ttlAttribute, region = 'us-east-1') =>
 {

    const client = new DynamoDBClient({
        region: region,
        endpoint: `https://dynamodb.${region}.amazonaws.com`
    });

    const params = {
        TableName: tableName,
        TimeToLiveSpecification: {
            Enabled: true,
            AttributeName: ttlAttribute
        }
    };

    try {
        const response = await client.send(new UpdateTimeToLiveCommand(params));
        if (response.$metadata.httpStatusCode === 200) {
            console.log(`TTL enabled successfully for table ${tableName}, using
 attribute name ${ttlAttribute}.`);
        } else {
            console.log(`Failed to enable TTL for table ${tableName}, response
 object: ${response}`);
        }
        return response;
    } catch (e) {
        console.error(`Error enabling TTL: ${e}`);
        throw e;
    }
};

// Example usage (commented out for testing)
// enableTTL('ExampleTable', 'exampleTtlAttribute');
```

Disable TTL on an existing DynamoDB table.

```
import { DynamoDBClient, UpdateTimeToLiveCommand } from "@aws-sdk/client-
dynamodb";

export const disableTTL = async (tableName, ttlAttribute, region = 'us-east-1')
 => {

    const client = new DynamoDBClient({
        region: region,
        endpoint: `https://dynamodb.${region}.amazonaws.com`
    });

    const params = {
        TableName: tableName,
        TimeToLiveSpecification: {
            Enabled: false,
            AttributeName: ttlAttribute
        }
    };

    try {
        const response = await client.send(new UpdateTimeToLiveCommand(params));
        if (response.$metadata.httpStatusCode === 200) {
            console.log(`TTL disabled successfully for table ${tableName}, using
 attribute name ${ttlAttribute}.`);
        } else {
            console.log(`Failed to disable TTL for table ${tableName}, response
 object: ${response}`);
        }
        return response;
    } catch (e) {
        console.error(`Error disabling TTL: ${e}`);
        throw e;
    }
};

// Example usage (commented out for testing)
// disableTTL('ExampleTable', 'exampleTtlAttribute');
```

- For API details, see UpdateTimeToLive in *AWS SDK for JavaScript API Reference*.

Python

SDK for Python (Boto3)

Enable TTL on an existing DynamoDB table.

```
import boto3


def enable_ttl(table_name, ttl_attribute_name):
    """
    Enables TTL on DynamoDB table for a given attribute name
        on success, returns a status code of 200
        on error, throws an exception

    :param table_name: Name of the DynamoDB table
    :param ttl_attribute_name: The name of the TTL attribute being provided to
 the table.
    """
    try:
        dynamodb = boto3.client("dynamodb")

        # Enable TTL on an existing DynamoDB table
        response = dynamodb.update_time_to_live(
            TableName=table_name,
            TimeToLiveSpecification={"Enabled": True, "AttributeName":
 ttl_attribute_name},
        )

        # In the returned response, check for a successful status code.
        if response["ResponseMetadata"]["HTTPStatusCode"] == 200:
            print("TTL has been enabled successfully.")
        else:
            print(
                f"Failed to enable TTL, status code {response['ResponseMetadata']
['HTTPStatusCode']}"
            )
        return response
    except Exception as ex:
        print("Couldn't enable TTL in table %s. Here's why: %s" % (table_name,
 ex))
        raise
```

```
# your values
enable_ttl("your-table-name", "expireAt")
```

Disable TTL on an existing DynamoDB table.

```
import boto3


def disable_ttl(table_name, ttl_attribute_name):
    """
    Disables TTL on DynamoDB table for a given attribute name
        on success, returns a status code of 200
        on error, throws an exception

    :param table_name: Name of the DynamoDB table being modified
    :param ttl_attribute_name: The name of the TTL attribute being provided to
 the table.
    """
    try:
        dynamodb = boto3.client("dynamodb")

        # Enable TTL on an existing DynamoDB table
        response = dynamodb.update_time_to_live(
            TableName=table_name,
            TimeToLiveSpecification={"Enabled": False, "AttributeName":
 ttl_attribute_name},
        )

        # In the returned response, check for a successful status code.
        if response["ResponseMetadata"]["HTTPStatusCode"] == 200:
            print("TTL has been disabled successfully.")
        else:
            print(
                f"Failed to disable TTL, status code
 {response['ResponseMetadata']['HTTPStatusCode']}"
            )
    except Exception as ex:
        print("Couldn't disable TTL in table %s. Here's why: %s" % (table_name,
 ex))
        raise
```

```
# your values
disable_ttl("your-table-name", "expireAt")
```

- For API details, see UpdateTimeToLive in *AWS SDK for Python (Boto3) API Reference*.

For a complete list of AWS SDK developer guides and code examples, see Using DynamoDB with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Scenarios for DynamoDB using AWS SDKs

The following code examples show you how to implement common scenarios in DynamoDB with AWS SDKs. These scenarios show you how to accomplish specific tasks by calling multiple functions within DynamoDB or combined with other AWS services. Each scenario includes a link to the complete source code, where you can find instructions on how to set up and run the code.

Scenarios target an intermediate level of experience to help you understand service actions in context.

Examples

- Accelerate DynamoDB reads with DAX using an AWS SDK
- Work with advanced DynamoDB Global Secondary Index scenarios using AWS Command Line Interface v2
- Build an application to submit data to a DynamoDB table
- Compare multiple values with a single attribute in DynamoDB with an AWS SDK
- Conditionally update a DynamoDB item with a TTL using an AWS SDK
- Connect to a local DynamoDB instance using an AWS SDK
- Count expression operators in DynamoDB with an AWS SDK
- Create an API Gateway REST API to track COVID-19 data
- Create a messenger application with Step Functions
- Create a photo asset management application that lets users manage photos using labels
- Create a DynamoDB table with a Global Secondary Index using the AWS SDK
- Create a DynamoDB table with warm throughput setting using an AWS SDK

- Create a web application to track DynamoDB data
- Create a websocket chat application with API Gateway
- Create a DynamoDB item with a TTL using an AWS SDK
- Create and manage DynamoDB global tables with Multi-Region Strong Consistency using an AWS SDK
- Create and manage DynamoDB global tables demonstrating MREC using an AWS SDK
- Delete DynamoDB data using PartiQL DELETE statements with an AWS SDK
- Detect PPE in images with Amazon Rekognition using an AWS SDK
- Insert DynamoDB data using PartiQL INSERT statements with an AWS SDK
- Invoke a Lambda function from a browser
- Manage DynamoDB Global Secondary Indexes using AWS Command Line Interface v2
- Manage DynamoDB resource-based policies using AWS Command Line Interface v2
- Monitor performance of Amazon DynamoDB using an AWS SDK
- Perform advanced DynamoDB query operations using an AWS SDK
- Perform list operations in DynamoDB with an AWS SDK
- Perform map operations in DynamoDB with an AWS SDK
- Perform set operations in DynamoDB with an AWS SDK
- Query a DynamoDB table by using batches of PartiQL statements and an AWS SDK
- Query a DynamoDB table using PartiQL and an AWS SDK
- Query a DynamoDB table using a Global Secondary Index with an AWS SDK
- Query a DynamoDB table using a begins_with condition with an AWS SDK
- Query a DynamoDB table using a date range in the sort key with an AWS SDK
- Query a DynamoDB table with a complex filter expression with an AWS SDK
- Query a DynamoDB table with a dynamic filter expression with an AWS SDK
- Query a DynamoDB table with a filter expression and limit with an AWS SDK
- Query a DynamoDB table with nested attributes using an AWS SDK
- Query a DynamoDB table with pagination using an AWS SDK
- Query a DynamoDB table with strongly consistent reads using an AWS SDK
- Query DynamoDB data using PartiQL SELECT statements with an AWS SDK

- Query a DynamoDB table for TTL items using an AWS SDK
- Query DynamoDB tables using date and time patterns with an AWS SDK
- Save EXIF and other image information using an AWS SDK
- Set up Attribute-Based Access Control for DynamoDB using AWS Command Line Interface v2
- Understand update expression order in DynamoDB with an AWS SDK
- Update a DynamoDB table setting with warm throughput using an AWS SDK
- Update a DynamoDB item with a TTL using an AWS SDK
- Update DynamoDB data using PartiQL UPDATE statements with an AWS SDK
- Use API Gateway to invoke a Lambda function
- Use Step Functions to invoke Lambda functions
- Use a document model for DynamoDB using an AWS SDK
- Use a high-level object persistence model for DynamoDB using an AWS SDK
- Use atomic counter operations in DynamoDB with an AWS SDK
- Use conditional operations in DynamoDB with an AWS SDK
- Use expression attribute names in DynamoDB with an AWS SDK
- Use scheduled events to invoke a Lambda function
- Work with DynamoDB Local Secondary Indexes using AWS Command Line Interface v2
- Work with DynamoDB Streams and Time-to-Live using AWS Command Line Interface v2
- Work with DynamoDB global tables and multi-Region replication with eventual consistency (MREC) using AWS Command Line Interface v2
- Work with DynamoDB resource tagging using AWS Command Line Interface v2
- Work with DynamoDB table encryption using AWS Command Line Interface v2

Accelerate DynamoDB reads with DAX using an AWS SDK

The following code example shows how to:

- Create and write data to a table with both the DAX and SDK clients.
- Get, query, and scan the table with both clients and compare their performance.

For more information, see Developing with the DynamoDB Accelerator Client.

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Create a table with either the DAX or Boto3 client.

```
import boto3


def create_dax_table(dyn_resource=None):
    """
    Creates a DynamoDB table.

    :param dyn_resource: Either a Boto3 or DAX resource.
    :return: The newly created table.
    """
    if dyn_resource is None:
        dyn_resource = boto3.resource("dynamodb")

    table_name = "TryDaxTable"
    params = {
        "TableName": table_name,
        "KeySchema": [
            {"AttributeName": "partition_key", "KeyType": "HASH"},
            {"AttributeName": "sort_key", "KeyType": "RANGE"},
        ],
        "AttributeDefinitions": [
            {"AttributeName": "partition_key", "AttributeType": "N"},
            {"AttributeName": "sort_key", "AttributeType": "N"},
        ],
        "BillingMode": "PAY_PER_REQUEST",
    }
    table = dyn_resource.create_table(**params)
    print(f"Creating {table_name}...")
    table.wait_until_exists()
    return table
```

```
if __name__ == "__main__":
    dax_table = create_dax_table()
    print(f"Created table.")
```

Write test data to the table.

```
import boto3


def write_data_to_dax_table(key_count, item_size, dyn_resource=None):
    """
    Writes test data to the demonstration table.

    :param key_count: The number of partition and sort keys to use to populate
 the
                      table. The total number of items is key_count * key_count.
    :param item_size: The size of non-key data for each test item.
    :param dyn_resource: Either a Boto3 or DAX resource.
    """
    if dyn_resource is None:
        dyn_resource = boto3.resource("dynamodb")

    table = dyn_resource.Table("TryDaxTable")
    some_data = "X" * item_size

    for partition_key in range(1, key_count + 1):
        for sort_key in range(1, key_count + 1):
            table.put_item(
                Item={
                    "partition_key": partition_key,
                    "sort_key": sort_key,
                    "some_data": some_data,
                }
            )
            print(f"Put item ({partition_key}, {sort_key}) succeeded.")


if __name__ == "__main__":
    write_key_count = 10
    write_item_size = 1000
    print(
```

```
        f"Writing {write_key_count*write_key_count} items to the table. "
        f"Each item is {write_item_size} characters."
    )
    write_data_to_dax_table(write_key_count, write_item_size)
```

Get items for a number of iterations for both the DAX client and the Boto3 client and report the time spent for each.

```
import argparse
import sys
import time
import amazondax
import boto3


def get_item_test(key_count, iterations, dyn_resource=None):
    """
    Gets items from the table a specified number of times. The time before the
    first iteration and the time after the last iteration are both captured
    and reported.

    :param key_count: The number of items to get from the table in each
 iteration.
    :param iterations: The number of iterations to run.
    :param dyn_resource: Either a Boto3 or DAX resource.
    :return: The start and end times of the test.
    """
    if dyn_resource is None:
        dyn_resource = boto3.resource("dynamodb")

    table = dyn_resource.Table("TryDaxTable")
    start = time.perf_counter()
    for _ in range(iterations):
        for partition_key in range(1, key_count + 1):
            for sort_key in range(1, key_count + 1):
                table.get_item(
                    Key={"partition_key": partition_key, "sort_key": sort_key}
                )
                print(".", end="")
                sys.stdout.flush()
    print()
    end = time.perf_counter()
```

```
    return start, end


if __name__ == "__main__":
    # pylint: disable=not-context-manager
    parser = argparse.ArgumentParser()
    parser.add_argument(
        "endpoint_url",
        nargs="?",
        help="When specified, the DAX cluster endpoint. Otherwise, DAX is not
 used.",
    )
    args = parser.parse_args()

    test_key_count = 10
    test_iterations = 50
    if args.endpoint_url:
        print(
            f"Getting each item from the table {test_iterations} times, "
            f"using the DAX client."
        )
        # Use a with statement so the DAX client closes the cluster after
 completion.
        with amazondax.AmazonDaxClient.resource(endpoint_url=args.endpoint_url)
 as dax:
            test_start, test_end = get_item_test(
                test_key_count, test_iterations, dyn_resource=dax
            )
    else:
        print(
            f"Getting each item from the table {test_iterations} times, "
            f"using the Boto3 client."
        )
        test_start, test_end = get_item_test(test_key_count, test_iterations)
    print(
        f"Total time: {test_end - test_start:.4f} sec. Average time: "
        f"{(test_end - test_start)/ test_iterations}."
    )
```

Query the table for a number of iterations for both the DAX client and the Boto3 client and report the time spent for each.

```
import argparse
import time
import sys
import amazondax
import boto3
from boto3.dynamodb.conditions import Key


def query_test(partition_key, sort_keys, iterations, dyn_resource=None):
    """
    Queries the table a specified number of times. The time before the
    first iteration and the time after the last iteration are both captured
    and reported.

    :param partition_key: The partition key value to use in the query. The query
                          returns items that have partition keys equal to this
 value.
    :param sort_keys: The range of sort key values for the query. The query
 returns
                      items that have sort key values between these two values.
    :param iterations: The number of iterations to run.
    :param dyn_resource: Either a Boto3 or DAX resource.
    :return: The start and end times of the test.
    """
    if dyn_resource is None:
        dyn_resource = boto3.resource("dynamodb")

    table = dyn_resource.Table("TryDaxTable")
    key_condition_expression = Key("partition_key").eq(partition_key) & Key(
        "sort_key"
    ).between(*sort_keys)

    start = time.perf_counter()
    for _ in range(iterations):
        table.query(KeyConditionExpression=key_condition_expression)
        print(".", end="")
        sys.stdout.flush()
    print()
    end = time.perf_counter()
    return start, end


if __name__ == "__main__":
```

```
    # pylint: disable=not-context-manager
    parser = argparse.ArgumentParser()
    parser.add_argument(
        "endpoint_url",
        nargs="?",
        help="When specified, the DAX cluster endpoint. Otherwise, DAX is not
used.",
    )
    args = parser.parse_args()

    test_partition_key = 5
    test_sort_keys = (2, 9)
    test_iterations = 100
    if args.endpoint_url:
        print(f"Querying the table {test_iterations} times, using the DAX
client.")
        # Use a with statement so the DAX client closes the cluster after
completion.
        with amazondax.AmazonDaxClient.resource(endpoint_url=args.endpoint_url)
as dax:
            test_start, test_end = query_test(
                test_partition_key, test_sort_keys, test_iterations,
dyn_resource=dax
            )
    else:
        print(f"Querying the table {test_iterations} times, using the Boto3
client.")
        test_start, test_end = query_test(
            test_partition_key, test_sort_keys, test_iterations
        )

    print(
        f"Total time: {test_end - test_start:.4f} sec. Average time: "
        f"{(test_end - test_start)/test_iterations}."
    )
```

Scan the table for a number of iterations for both the DAX client and the Boto3 client and report the time spent for each.

```
import argparse
import time
import sys
```

```
import amazondax
import boto3


def scan_test(iterations, dyn_resource=None):
    """
    Scans the table a specified number of times. The time before the
    first iteration and the time after the last iteration are both captured
    and reported.

    :param iterations: The number of iterations to run.
    :param dyn_resource: Either a Boto3 or DAX resource.
    :return: The start and end times of the test.
    """
    if dyn_resource is None:
        dyn_resource = boto3.resource("dynamodb")

    table = dyn_resource.Table("TryDaxTable")
    start = time.perf_counter()
    for _ in range(iterations):
        table.scan()
        print(".", end="")
        sys.stdout.flush()
    print()
    end = time.perf_counter()
    return start, end


if __name__ == "__main__":
    # pylint: disable=not-context-manager
    parser = argparse.ArgumentParser()
    parser.add_argument(
        "endpoint_url",
        nargs="?",
        help="When specified, the DAX cluster endpoint. Otherwise, DAX is not
 used.",
    )
    args = parser.parse_args()

    test_iterations = 100
    if args.endpoint_url:
        print(f"Scanning the table {test_iterations} times, using the DAX
 client.")
```

```
        # Use a with statement so the DAX client closes the cluster after
 completion.
        with amazondax.AmazonDaxClient.resource(endpoint_url=args.endpoint_url)
 as dax:
            test_start, test_end = scan_test(test_iterations, dyn_resource=dax)
    else:
        print(f"Scanning the table {test_iterations} times, using the Boto3
 client.")
        test_start, test_end = scan_test(test_iterations)
    print(
        f"Total time: {test_end - test_start:.4f} sec. Average time: "
        f"{(test_end - test_start)/test_iterations}."
    )
```

Delete the table.

```
import boto3


def delete_dax_table(dyn_resource=None):
    """
    Deletes the demonstration table.

    :param dyn_resource: Either a Boto3 or DAX resource.
    """
    if dyn_resource is None:
        dyn_resource = boto3.resource("dynamodb")

    table = dyn_resource.Table("TryDaxTable")
    table.delete()

    print(f"Deleting {table.name}...")
    table.wait_until_not_exists()


if __name__ == "__main__":
    delete_dax_table()
    print("Table deleted!")
```

- For API details, see the following topics in *AWS SDK for Python (Boto3) API Reference*.
  - CreateTable

- DeleteTable
- GetItem
- PutItem
- Query
- Scan

For a complete list of AWS SDK developer guides and code examples, see Using DynamoDB with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Work with advanced DynamoDB Global Secondary Index scenarios using AWS Command Line Interface v2

The following code example shows how to work with advanced Global Secondary Index configurations.

- Create a table with multiple GSIs.
- Create a table with on-demand capacity and GSI.
- Put items into a table with multiple GSIs.
- Query multiple GSIs with different conditions.

Bash

AWS CLI with Bash script

Create a table with multiple GSIs.

```
# Create a table with multiple GSIs
aws dynamodb create-table \
    --table-name MusicLibrary \
    --attribute-definitions \
        AttributeName=Artist,AttributeType=S \
        AttributeName=SongTitle,AttributeType=S \
        AttributeName=AlbumTitle,AttributeType=S \
        AttributeName=Genre,AttributeType=S \
        AttributeName=Year,AttributeType=N \
    --key-schema \
```

```
        AttributeName=Artist,KeyType=HASH \
        AttributeName=SongTitle,KeyType=RANGE \
    --billing-mode PAY_PER_REQUEST \
    --global-secondary-indexes \
        "[
            {
                \"IndexName\": \"AlbumIndex\",
                \"KeySchema\": [{\"AttributeName\":\"AlbumTitle\",\"KeyType\":
\"HASH\"}],
                \"Projection\": {\"ProjectionType\":\"ALL\"}
            },
            {
                \"IndexName\": \"GenreYearIndex\",
                \"KeySchema\": [
                    {\"AttributeName\":\"Genre\",\"KeyType\":\"HASH\"},
                    {\"AttributeName\":\"Year\",\"KeyType\":\"RANGE\"}
                ],
                \"Projection\": {\"ProjectionType\":\"INCLUDE\",
\"NonKeyAttributes\":[\"Artist\",\"SongTitle\"]}
            }
        ]"
```

Create a table with on-demand capacity and GSI.

```
# Create a table with on-demand capacity and GSI
aws dynamodb create-table \
    --table-name MusicOnDemand \
    --attribute-definitions \
        AttributeName=Artist,AttributeType=S \
        AttributeName=SongTitle,AttributeType=S \
        AttributeName=Genre,AttributeType=S \
    --key-schema \
        AttributeName=Artist,KeyType=HASH \
        AttributeName=SongTitle,KeyType=RANGE \
    --billing-mode PAY_PER_REQUEST \
    --global-secondary-indexes \
        "[
            {
                \"IndexName\": \"GenreIndex\",
                \"KeySchema\": [{\"AttributeName\":\"Genre\",\"KeyType\":\"HASH
\"}],
                \"Projection\": {\"ProjectionType\":\"ALL\"}
```

```
            }
        ]"
```

Put items into a table with multiple GSIs.

```
# Add items to MusicLibrary table
aws dynamodb put-item \
    --table-name MusicLibrary \
    --item '{
        "Artist": {"S": "The Beatles"},
        "SongTitle": {"S": "Hey Jude"},
        "AlbumTitle": {"S": "Past Masters"},
        "Genre": {"S": "Rock"},
        "Year": {"N": "1968"}
    }'

aws dynamodb put-item \
    --table-name MusicLibrary \
    --item '{
        "Artist": {"S": "Miles Davis"},
        "SongTitle": {"S": "So What"},
        "AlbumTitle": {"S": "Kind of Blue"},
        "Genre": {"S": "Jazz"},
        "Year": {"N": "1959"}
    }'
```

Query items from a table with multiple GSIs.

```
# Query the AlbumIndex GSI
echo "Querying AlbumIndex GSI:"
aws dynamodb query \
    --table-name MusicLibrary \
    --index-name AlbumIndex \
    --key-condition-expression "AlbumTitle = :album" \
    --expression-attribute-values '{":album":{"S":"Kind of Blue"}}'

# Query the GenreYearIndex GSI with a range condition
echo "Querying GenreYearIndex GSI with range condition:"
aws dynamodb query \
    --table-name MusicLibrary \
    --index-name GenreYearIndex \
```

```
    --key-condition-expression "Genre = :genre AND #yr > :year" \
    --expression-attribute-names '{"#yr": "Year"}' \
    --expression-attribute-values '{":genre":{"S":"Rock"},":year":{"N":"1965"}}'
```

- For API details, see the following topics in *AWS CLI Command Reference*.
  - CreateTable
  - PutItem
  - Query

For a complete list of AWS SDK developer guides and code examples, see Using DynamoDB with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Build an application to submit data to a DynamoDB table

The following code examples show how to build an application that submits data to an Amazon DynamoDB table and notifies you when a user updates the table.

Java

SDK for Java 2.x

Shows how to create a dynamic web application that submits data using the Amazon DynamoDB Java API and sends a text message using the Amazon Simple Notification Service Java API.

For complete source code and instructions on how to set up and run, see the full example on GitHub.

Services used in this example

- DynamoDB
- Amazon SNS

JavaScript

SDK for JavaScript (v3)

This example shows how to build an app that enables users to submit data to an Amazon DynamoDB table, and send a text message to the administrator using Amazon Simple Notification Service (Amazon SNS).

For complete source code and instructions on how to set up and run, see the full example on GitHub.

This example is also available in the AWS SDK for JavaScript v3 developer guide.

Services used in this example

- DynamoDB
- Amazon SNS

For a complete list of AWS SDK developer guides and code examples, see Using DynamoDB with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Compare multiple values with a single attribute in DynamoDB with an AWS SDK

The following code examples show how to compare multiple values with a single attribute in DynamoDB.

- Use the IN operator to compare multiple values with a single attribute.
- Compare the IN operator with multiple OR conditions.
- Understand the performance and expression complexity benefits of using IN.

Java

SDK for Java 2.x

Compare multiple values with a single attribute in DynamoDB using AWS SDK for Java 2.x.

```
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
```

```
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.QueryRequest;
import software.amazon.awssdk.services.dynamodb.model.QueryResponse;
import software.amazon.awssdk.services.dynamodb.model.ScanRequest;
import software.amazon.awssdk.services.dynamodb.model.ScanResponse;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Locale;
import java.util.Map;

    /**
     * Queries a table using the IN operator to compare multiple values with a
 single attribute.
     *
     * <p>This method demonstrates how to use the IN operator in a filter
 expression
     * to match an attribute against multiple values.
     *
     * @param dynamoDbClient The DynamoDB client
     * @param tableName The name of the DynamoDB table
     * @param partitionKeyName The name of the partition key attribute
     * @param partitionKeyValue The value of the partition key to query
     * @param attributeName The name of the attribute to compare
     * @param valuesList List of values to compare against
     * @return The query response from DynamoDB
     * @throws DynamoDbException if an error occurs during the operation
     */
    public static QueryResponse compareMultipleValues(
        DynamoDbClient dynamoDbClient,
        String tableName,
        String partitionKeyName,
        AttributeValue partitionKeyValue,
        String attributeName,
        List<AttributeValue> valuesList) {

        // Create expression attribute names
        Map<String, String> expressionAttributeNames = new HashMap<>();
        expressionAttributeNames.put("#pkName", partitionKeyName);
        expressionAttributeNames.put("#attrName", attributeName);

        // Create expression attribute values
        Map<String, AttributeValue> expressionAttributeValues = new HashMap<>();
```

```
        expressionAttributeValues.put(":pkValue", partitionKeyValue);

        // Add values for IN operator
        for (int i = 0; i < valuesList.size(); i++) {
            expressionAttributeValues.put(":val" + i, valuesList.get(i));
        }

        // Build the IN clause
        StringBuilder inClause = new StringBuilder();
        for (int i = 0; i < valuesList.size(); i++) {
            if (i > 0) {
                inClause.append(", ");
            }
            inClause.append(":val").append(i);
        }

        // Define the query parameters
        QueryRequest request = QueryRequest.builder()
            .tableName(tableName)
            .keyConditionExpression("#pkName = :pkValue")
            .filterExpression("#attrName IN (" + inClause.toString() + ")")
            .expressionAttributeNames(expressionAttributeNames)
            .expressionAttributeValues(expressionAttributeValues)
            .build();

        // Perform the query operation
        return dynamoDbClient.query(request);
    }

    /**
     * Queries a table using multiple OR conditions to compare multiple values
with a single attribute.
     *
     * <p>This method demonstrates the alternative approach to using the IN
operator,
     * by using multiple OR conditions.
     *
     * @param dynamoDbClient The DynamoDB client
     * @param tableName The name of the DynamoDB table
     * @param partitionKeyName The name of the partition key attribute
     * @param partitionKeyValue The value of the partition key to query
     * @param attributeName The name of the attribute to compare
     * @param valuesList List of values to compare against
     * @return The query response from DynamoDB
```

```
     * @throws DynamoDbException if an error occurs during the operation
     */
    public static QueryResponse compareWithOrConditions(
        DynamoDbClient dynamoDbClient,
        String tableName,
        String partitionKeyName,
        AttributeValue partitionKeyValue,
        String attributeName,
        List<AttributeValue> valuesList) {

        // Create expression attribute names
        Map<String, String> expressionAttributeNames = new HashMap<>();
        expressionAttributeNames.put("#pkName", partitionKeyName);
        expressionAttributeNames.put("#attrName", attributeName);

        // Create expression attribute values
        Map<String, AttributeValue> expressionAttributeValues = new HashMap<>();
        expressionAttributeValues.put(":pkValue", partitionKeyValue);

        // Add values for OR conditions
        for (int i = 0; i < valuesList.size(); i++) {
            expressionAttributeValues.put(":val" + i, valuesList.get(i));
        }

        // Build the OR conditions
        StringBuilder orConditions = new StringBuilder();
        for (int i = 0; i < valuesList.size(); i++) {
            if (i > 0) {
                orConditions.append(" OR ");
            }
            orConditions.append("#attrName = :val").append(i);
        }

        // Define the query parameters
        QueryRequest request = QueryRequest.builder()
            .tableName(tableName)
            .keyConditionExpression("#pkName = :pkValue")
            .filterExpression(orConditions.toString())
            .expressionAttributeNames(expressionAttributeNames)
            .expressionAttributeValues(expressionAttributeValues)
            .build();

        // Perform the query operation
        return dynamoDbClient.query(request);
```

```
    }

    /**
     * Compares the performance of using the IN operator versus multiple OR
conditions.
     *
     * <p>This method demonstrates the performance difference between using the
IN operator
     * and using multiple OR conditions.
     *
     * @param dynamoDbClient The DynamoDB client
     * @param tableName The name of the DynamoDB table
     * @param partitionKeyName The name of the partition key attribute
     * @param partitionKeyValue The value of the partition key to query
     * @param attributeName The name of the attribute to compare
     * @param valuesList List of values to compare against
     * @return Map containing the performance comparison results
     */
    public static Map<String, Object> comparePerformance(
        DynamoDbClient dynamoDbClient,
        String tableName,
        String partitionKeyName,
        AttributeValue partitionKeyValue,
        String attributeName,
        List<AttributeValue> valuesList) {

        Map<String, Object> results = new HashMap<>();

        try {
            // Measure performance of IN operator
            long inStartTime = System.nanoTime();
            QueryResponse inResponse = compareMultipleValues(
                dynamoDbClient, tableName, partitionKeyName, partitionKeyValue,
attributeName, valuesList);
            long inEndTime = System.nanoTime();
            long inDuration = inEndTime - inStartTime;

            // Measure performance of OR conditions
            long orStartTime = System.nanoTime();
            QueryResponse orResponse = compareWithOrConditions(
                dynamoDbClient, tableName, partitionKeyName, partitionKeyValue,
attributeName, valuesList);
            long orEndTime = System.nanoTime();
            long orDuration = orEndTime - orStartTime;
```

```
            // Record results
            results.put("inOperatorDuration", inDuration);
            results.put("orConditionsDuration", orDuration);
            results.put("inOperatorItems", inResponse.count());
            results.put("orConditionsItems", orResponse.count());
            results.put("inOperatorExpression", "IN operator with " +
valuesList.size() + " values");
            results.put("orConditionsExpression", valuesList.size() + " OR
conditions");
            results.put("success", true);

        } catch (DynamoDbException e) {
            results.put("success", false);
            results.put("error", e.getMessage());
        }

        return results;
    }

    /**
     * Scans a table using the IN operator with a large number of values.
     *
     * <p>This method demonstrates how to use the IN operator with a large number
of values,
     * which can help stay within the 300 operator limit.
     *
     * @param dynamoDbClient The DynamoDB client
     * @param tableName The name of the DynamoDB table
     * @param attributeName The name of the attribute to compare
     * @param valuesList List of values to compare against
     * @return The scan response from DynamoDB
     * @throws DynamoDbException if an error occurs during the operation
     */
    public static ScanResponse scanWithLargeInClause(
        DynamoDbClient dynamoDbClient, String tableName, String attributeName,
List<AttributeValue> valuesList) {

        // Create expression attribute names
        Map<String, String> expressionAttributeNames = new HashMap<>();
        expressionAttributeNames.put("#attrName", attributeName);

        // Create expression attribute values
        Map<String, AttributeValue> expressionAttributeValues = new HashMap<>();
```

```
        // Add values for IN operator
        for (int i = 0; i < valuesList.size(); i++) {
            expressionAttributeValues.put(":val" + i, valuesList.get(i));
        }

        // Build the IN clause
        StringBuilder inClause = new StringBuilder();
        for (int i = 0; i < valuesList.size(); i++) {
            if (i > 0) {
                inClause.append(", ");
            }
            inClause.append(":val").append(i);
        }

        // Define the scan parameters
        ScanRequest request = ScanRequest.builder()
            .tableName(tableName)
            .filterExpression("#attrName IN (" + inClause.toString() + ")")
            .expressionAttributeNames(expressionAttributeNames)
            .expressionAttributeValues(expressionAttributeValues)
            .build();

        // Perform the scan operation
        return dynamoDbClient.scan(request);
    }

    /**
     * Generates a list of sample values for testing.
     *
     * <p>Helper method to generate a list of sample values for testing.
     *
     * @param valueType The type of values to generate (string, number, or
boolean)
     * @param count The number of values to generate
     * @return List of generated attribute values
     */
    public static List<AttributeValue> generateSampleValues(String valueType, int
count) {
        List<AttributeValue> values = new ArrayList<>();

        for (int i = 0; i < count; i++) {
            AttributeValue value;
```

```
            switch (valueType.toLowerCase(Locale.ROOT)) {
                case "string":
                    value = AttributeValue.builder().s("Value" + i).build();
                    break;
                case "number":
                    value =
AttributeValue.builder().n(String.valueOf(i)).build();
                    break;
                case "boolean":
                    value = AttributeValue.builder().bool(i % 2 == 0).build();
                    break;
                default:
                    throw new IllegalArgumentException("Unsupported value type: "
+ valueType);
            }

            values.add(value);
        }

        return values;
    }
```

Example usage of comparing multiple values with AWS SDK for Java 2.x.

```
    public static void exampleUsage(DynamoDbClient dynamoDbClient, String
tableName) {
        System.out.println("Demonstrating how to compare multiple values with a
single attribute in DynamoDB");

        try {
            // Example 1: Using the IN operator
            System.out.println("\nExample 1: Using the IN operator");
            List<AttributeValue> categories = List.of(
                AttributeValue.builder().s("Electronics").build(),
                AttributeValue.builder().s("Computers").build(),
                AttributeValue.builder().s("Accessories").build());

            QueryResponse inResponse = compareMultipleValues(
                dynamoDbClient,
                tableName,
                "Department",
                AttributeValue.builder().s("Retail").build(),
```

```
                "Category",
                categories);

            System.out.println("Found " + inResponse.count() + " items using IN
operator");
            System.out.println("Items: " + inResponse.items());

            // Example 2: Using multiple OR conditions
            System.out.println("\nExample 2: Using multiple OR conditions");
            QueryResponse orResponse = compareWithOrConditions(
                dynamoDbClient,
                tableName,
                "Department",
                AttributeValue.builder().s("Retail").build(),
                "Category",
                categories);

            System.out.println("Found " + orResponse.count() + " items using OR
conditions");
            System.out.println("Items: " + orResponse.items());

            // Example 3: Performance comparison
            System.out.println("\nExample 3: Performance comparison");
            Map<String, Object> perfComparison = comparePerformance(
                dynamoDbClient,
                tableName,
                "Department",
                AttributeValue.builder().s("Retail").build(),
                "Category",
                categories);

            if ((boolean) perfComparison.get("success")) {
                System.out.println("IN operator duration: " +
perfComparison.get("inOperatorDuration") + " ns");
                System.out.println("OR conditions duration: " +
perfComparison.get("orConditionsDuration") + " ns");
                System.out.println("IN operator found " +
perfComparison.get("inOperatorItems") + " items");
                System.out.println("OR conditions found " +
perfComparison.get("orConditionsItems") + " items");
                System.out.println("Expression complexity comparison:");
                System.out.println("  IN operator: " +
perfComparison.get("inOperatorExpression"));
```

```
                System.out.println("  OR conditions: " +
perfComparison.get("orConditionsExpression"));
            } else {
                System.out.println("Performance comparison failed: " +
perfComparison.get("error"));
            }

            // Example 4: Using IN with a large number of values
            System.out.println("\nExample 4: Using IN with a large number of
values");
            List<AttributeValue> productIds = generateSampleValues("string", 20);

            ScanResponse largeInResponse = scanWithLargeInClause(dynamoDbClient,
tableName, "ProductId", productIds);

            System.out.println(
                "Found " + largeInResponse.count() + " items using IN with " +
productIds.size() + " values");

            // Explain the benefits of using IN
            System.out.println("\nKey points about using the IN operator in
DynamoDB:");
            System.out.println("1. The IN operator allows comparing a single
attribute against multiple values");
            System.out.println("2. IN is more concise than using multiple OR
conditions");
            System.out.println("3. IN counts as only 1 operator regardless of the
number of values");
            System.out.println("4. Multiple OR conditions count as 1 operator per
condition plus 1 per OR");
            System.out.println("5. Using IN helps stay within the 300 operator
limit for complex expressions");
            System.out.println("6. IN can be used in filter expressions and
condition expressions");
            System.out.println("7. The IN operator supports up to 100 comparison
values");

        } catch (DynamoDbException e) {
            System.err.println("Error: " + e.getMessage());
            e.printStackTrace();
        }
    }
```

- For API details, see the following topics in *AWS SDK for Java 2.x API Reference*.
  - Query
  - Scan

JavaScript

SDK for JavaScript (v3)

Compare multiple values with a single attribute using AWS SDK for JavaScript.

```
const { DynamoDBClient } = require("@aws-sdk/client-dynamodb");
const {
  DynamoDBDocumentClient,
  ScanCommand,
  QueryCommand
} = require("@aws-sdk/lib-dynamodb");

/**
 * Query or scan a DynamoDB table to find items where an attribute matches any
 value from a list.
 *
 * This function demonstrates the use of the IN operator to compare a single
 attribute
 * against multiple possible values, which is more efficient than using multiple
 OR conditions.
 *
 * @param {Object} config - AWS configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {string} attributeName - The name of the attribute to compare against
 the values list
 * @param {Array} valuesList - List of values to compare the attribute against
 * @param {string} [partitionKeyName] - Optional name of the partition key
 attribute for query operations
 * @param {string} [partitionKeyValue] - Optional value of the partition key to
 query
 * @returns {Promise<Object>} - The response from DynamoDB containing the
 matching items
 */
async function compareMultipleValues(
  config,
  tableName,
  attributeName,
```

```
  valuesList,
  partitionKeyName,
  partitionKeyValue
) {
  // Initialize the DynamoDB client
  const client = new DynamoDBClient(config);
  const docClient = DynamoDBDocumentClient.from(client);

  // Create the filter expression using the IN operator
  const filterExpression = `${attributeName} IN (${valuesList.map((_, index) =>
 `:val${index}`).join(', ')})`;

  // Create expression attribute values for the values list
  const expressionAttributeValues = valuesList.reduce((acc, val, index) => {
    acc[`:val${index}`] = val;
    return acc;
  }, {});

  // If partition key is provided, perform a query operation
  if (partitionKeyName && partitionKeyValue) {
    const keyCondition = `${partitionKeyName} = :partitionKey`;
    expressionAttributeValues[':partitionKey'] = partitionKeyValue;

    // Initialize array to collect all items
    let allItems = [];
    let lastEvaluatedKey;

    // Use pagination to get all results
    do {
      const params = {
        TableName: tableName,
        KeyConditionExpression: keyCondition,
        FilterExpression: filterExpression,
        ExpressionAttributeValues: expressionAttributeValues
      };

      // Add ExclusiveStartKey if we have a lastEvaluatedKey from a previous
 query
      if (lastEvaluatedKey) {
        params.ExclusiveStartKey = lastEvaluatedKey;
      }

      const response = await docClient.send(new QueryCommand(params));
```

```
      // Add the items from this page to our collection
      if (response.Items && response.Items.length > 0) {
        allItems = [...allItems, ...response.Items];
      }

      // Get the key for the next page of results
      lastEvaluatedKey = response.LastEvaluatedKey;
    } while (lastEvaluatedKey);

    // Return the complete result
    return {
      Items: allItems,
      Count: allItems.length
    };
  } else {
    // Otherwise, perform a scan operation
    // Initialize array to collect all items
    let allItems = [];
    let lastEvaluatedKey;

    // Use pagination to get all results
    do {
      const params = {
        TableName: tableName,
        FilterExpression: filterExpression,
        ExpressionAttributeValues: expressionAttributeValues
      };

      // Add ExclusiveStartKey if we have a lastEvaluatedKey from a previous scan
      if (lastEvaluatedKey) {
        params.ExclusiveStartKey = lastEvaluatedKey;
      }

      const response = await docClient.send(new ScanCommand(params));

      // Add the items from this page to our collection
      if (response.Items && response.Items.length > 0) {
        allItems = [...allItems, ...response.Items];
      }

      // Get the key for the next page of results
      lastEvaluatedKey = response.LastEvaluatedKey;
    } while (lastEvaluatedKey);
```

```
    // Return the complete result
    return {
      Items: allItems,
      Count: allItems.length
    };
  }
}

/**
 * Alternative implementation using multiple OR conditions instead of the IN
 operator.
 *
 * This function is provided for comparison to show why using the IN operator is
 preferable.
 * With many values, this approach becomes verbose and less efficient.
 *
 * @param {Object} config - AWS configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {string} attributeName - The name of the attribute to compare against
 the values list
 * @param {Array} valuesList - List of values to compare the attribute against
 * @param {string} [partitionKeyName] - Optional name of the partition key
 attribute for query operations
 * @param {string} [partitionKeyValue] - Optional value of the partition key to
 query
 * @returns {Promise<Object>} - The response from DynamoDB containing the
 matching items
 */
async function compareWithOrConditions(
  config,
  tableName,
  attributeName,
  valuesList,
  partitionKeyName,
  partitionKeyValue
) {
  // Initialize the DynamoDB client
  const client = new DynamoDBClient(config);
  const docClient = DynamoDBDocumentClient.from(client);

  // If no values provided, return empty result
  if (!valuesList || valuesList.length === 0) {
    return {
      Items: [],
```

```
      Count: 0
    };
  }

  // Create the filter expression using multiple OR conditions
  const filterConditions = valuesList.map((_, index) => `${attributeName} = :val
${index}`);
  const filterExpression = filterConditions.join(' OR ');

  // Create expression attribute values for the values list
  const expressionAttributeValues = valuesList.reduce((acc, val, index) => {
    acc[`:val${index}`] = val;
    return acc;
  }, {});

  // If partition key is provided, perform a query operation
  if (partitionKeyName && partitionKeyValue) {
    const keyCondition = `${partitionKeyName} = :partitionKey`;
    expressionAttributeValues[':partitionKey'] = partitionKeyValue;

    // Initialize array to collect all items
    let allItems = [];
    let lastEvaluatedKey;

    // Use pagination to get all results
    do {
      const params = {
        TableName: tableName,
        KeyConditionExpression: keyCondition,
        FilterExpression: filterExpression,
        ExpressionAttributeValues: expressionAttributeValues
      };

      // Add ExclusiveStartKey if we have a lastEvaluatedKey from a previous
 query
      if (lastEvaluatedKey) {
        params.ExclusiveStartKey = lastEvaluatedKey;
      }

      const response = await docClient.send(new QueryCommand(params));

      // Add the items from this page to our collection
      if (response.Items && response.Items.length > 0) {
        allItems = [...allItems, ...response.Items];
```

```
      }

      // Get the key for the next page of results
      lastEvaluatedKey = response.LastEvaluatedKey;
    } while (lastEvaluatedKey);

    // Return the complete result
    return {
      Items: allItems,
      Count: allItems.length
    };
  } else {
    // Otherwise, perform a scan operation
    // Initialize array to collect all items
    let allItems = [];
    let lastEvaluatedKey;

    // Use pagination to get all results
    do {
      const params = {
        TableName: tableName,
        FilterExpression: filterExpression,
        ExpressionAttributeValues: expressionAttributeValues
      };

      // Add ExclusiveStartKey if we have a lastEvaluatedKey from a previous scan
      if (lastEvaluatedKey) {
        params.ExclusiveStartKey = lastEvaluatedKey;
      }

      const response = await docClient.send(new ScanCommand(params));

      // Add the items from this page to our collection
      if (response.Items && response.Items.length > 0) {
        allItems = [...allItems, ...response.Items];
      }

      // Get the key for the next page of results
      lastEvaluatedKey = response.LastEvaluatedKey;
    } while (lastEvaluatedKey);

    // Return the complete result
    return {
      Items: allItems,
```

```
      Count: allItems.length
    };
  }
}
```

Example usage of comparing multiple values with AWS SDK for JavaScript.

```
/**
 * Example of how to use the compareMultipleValues function.
 */
async function exampleUsage() {
  // Example parameters
  const config = { region: "us-west-2" };
  const tableName = "Products";
  const attributeName = "Category";
  const valuesList = ["Electronics", "Computers", "Accessories"];

  console.log(`Searching for products in any of these categories:
 ${valuesList.join(', ')}`);

  try {
    // Using the IN operator (recommended approach)
    console.log("\nApproach 1: Using the IN operator");
    const response = await compareMultipleValues(
      config,
      tableName,
      attributeName,
      valuesList
    );

    console.log(`Found ${response.Count} products in the specified categories`);

    // Using multiple OR conditions (alternative approach)
    console.log("\nApproach 2: Using multiple OR conditions");
    const response2 = await compareWithOrConditions(
      config,
      tableName,
      attributeName,
      valuesList
    );

    console.log(`Found ${response2.Count} products in the specified categories`);
```

```
    // Example with a query operation
    console.log("\nQuerying a specific manufacturer's products in multiple
 categories");
    const partitionKeyName = "Manufacturer";
    const partitionKeyValue = "Acme";

    const response3 = await compareMultipleValues(
      config,
      tableName,
      attributeName,
      valuesList,
      partitionKeyName,
      partitionKeyValue
    );

    console.log(`Found ${response3.Count} Acme products in the specified
 categories`);

    // Explain the benefits of using the IN operator
    console.log("\nBenefits of using the IN operator:");
    console.log("1. More concise expression compared to multiple OR conditions");
    console.log("2. Better readability and maintainability");
    console.log("3. Potentially better performance with large value lists");
    console.log("4. Simpler code that's less prone to errors");
    console.log("5. Easier to modify when adding or removing values");

  } catch (error) {
    console.error("Error:", error);
  }
}
```

- For API details, see the following topics in *AWS SDK for JavaScript API Reference*.
  - Query
  - Scan

Python

SDK for Python (Boto3)

Compare multiple values with a single attribute using AWS SDK for Python (Boto3).

```
import boto3
from boto3.dynamodb.conditions import Attr, Key
from typing import Any, Dict, List, Optional


def compare_multiple_values(
    table_name: str,
    attribute_name: str,
    values_list: List[Any],
    partition_key_name: Optional[str] = None,
    partition_key_value: Optional[str] = None,
) -> Dict[str, Any]:
    """
    Query or scan a DynamoDB table to find items where an attribute matches any
 value from a list.

    This function demonstrates the use of the IN operator to compare a single
 attribute
    against multiple possible values, which is more efficient than using multiple
 OR conditions.

    Args:
        table_name (str): The name of the DynamoDB table.
        attribute_name (str): The name of the attribute to compare against the
 values list.
        values_list (List[Any]): List of values to compare the attribute against.
        partition_key_name (Optional[str]): The name of the partition key
 attribute for query operations.
        partition_key_value (Optional[str]): The value of the partition key to
 query.

    Returns:
        Dict[str, Any]: The response from DynamoDB containing the matching items.
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # Create the filter expression using the is_in method
    filter_expression = Attr(attribute_name).is_in(values_list)

    # If partition key is provided, perform a query operation
    if partition_key_name and partition_key_value:
```

```
        key_condition = Key(partition_key_name).eq(partition_key_value)
        response = table.query(
            KeyConditionExpression=key_condition,
 FilterExpression=filter_expression
        )
    else:
        # Otherwise, perform a scan operation
        response = table.scan(FilterExpression=filter_expression)

    # Handle pagination if there are more results
    items = response.get("Items", [])
    while "LastEvaluatedKey" in response:
        if partition_key_name and partition_key_value:
            response = table.query(
                KeyConditionExpression=key_condition,
                FilterExpression=filter_expression,
                ExclusiveStartKey=response["LastEvaluatedKey"],
            )
        else:
            response = table.scan(
                FilterExpression=filter_expression,
 ExclusiveStartKey=response["LastEvaluatedKey"]
            )
        items.extend(response.get("Items", []))

    # Return the complete result
    return {"Items": items, "Count": len(items)}


def compare_with_or_conditions(
    table_name: str,
    attribute_name: str,
    values_list: List[Any],
    partition_key_name: Optional[str] = None,
    partition_key_value: Optional[str] = None,
) -> Dict[str, Any]:
    """
    Alternative implementation using multiple OR conditions instead of the IN
 operator.

    This function is provided for comparison to show why using the IN operator is
 preferable.
    With many values, this approach becomes verbose and less efficient.
```

```
    Args:
        table_name (str): The name of the DynamoDB table.
        attribute_name (str): The name of the attribute to compare against the
values list.
        values_list (List[Any]): List of values to compare the attribute against.
        partition_key_name (Optional[str]): The name of the partition key
attribute for query operations.
        partition_key_value (Optional[str]): The value of the partition key to
query.

    Returns:
        Dict[str, Any]: The response from DynamoDB containing the matching items.
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # Create a filter expression with multiple OR conditions
    filter_expression = None
    for value in values_list:
        condition = Attr(attribute_name).eq(value)
        if filter_expression is None:
            filter_expression = condition
        else:
            filter_expression = filter_expression | condition

    # If partition key is provided, perform a query operation
    if partition_key_name and partition_key_value and filter_expression:
        key_condition = Key(partition_key_name).eq(partition_key_value)
        response = table.query(
            KeyConditionExpression=key_condition,
FilterExpression=filter_expression
        )
    elif filter_expression:
        # Otherwise, perform a scan operation
        response = table.scan(FilterExpression=filter_expression)
    else:
        # Return empty response if no values provided
        return {"Items": [], "Count": 0}

    # Handle pagination if there are more results
    items = response.get("Items", [])
    while "LastEvaluatedKey" in response:
        if partition_key_name and partition_key_value:
```

```
            response = table.query(
                KeyConditionExpression=key_condition,
                FilterExpression=filter_expression,
                ExclusiveStartKey=response["LastEvaluatedKey"],
            )
        else:
            response = table.scan(
                FilterExpression=filter_expression,
 ExclusiveStartKey=response["LastEvaluatedKey"]
            )
        items.extend(response.get("Items", []))

    # Return the complete result
    return {"Items": items, "Count": len(items)}
```

Example usage of comparing multiple values with AWS SDK for Python (Boto3).

```
def example_usage():
    """Example of how to use the compare_multiple_values function."""
    # Example parameters
    table_name = "Products"
    attribute_name = "Category"
    values_list = ["Electronics", "Computers", "Accessories"]

    print(f"Searching for products in any of these categories: {values_list}")

    # Using the IN operator (recommended approach)
    print("\nApproach 1: Using the IN operator")
    response = compare_multiple_values(
        table_name=table_name, attribute_name=attribute_name,
 values_list=values_list
    )

    print(f"Found {response['Count']} products in the specified categories")

    # Using multiple OR conditions (alternative approach)
    print("\nApproach 2: Using multiple OR conditions")
    response2 = compare_with_or_conditions(
        table_name=table_name, attribute_name=attribute_name,
 values_list=values_list
```

```
    )

    print(f"Found {response2['Count']} products in the specified categories")

    # Example with a query operation
    print("\nQuerying a specific manufacturer's products in multiple categories")
    partition_key_name = "Manufacturer"
    partition_key_value = "Acme"

    response3 = compare_multiple_values(
        table_name=table_name,
        attribute_name=attribute_name,
        values_list=values_list,
        partition_key_name=partition_key_name,
        partition_key_value=partition_key_value,
    )

    print(f"Found {response3['Count']} Acme products in the specified
categories")

    # Explain the benefits of using the IN operator
    print("\nBenefits of using the IN operator:")
    print("1. More concise expression compared to multiple OR conditions")
    print("2. Better readability and maintainability")
    print("3. Potentially better performance with large value lists")
    print("4. Simpler code that's less prone to errors")
    print("5. Easier to modify when adding or removing values")
```

- For API details, see the following topics in *AWS SDK for Python (Boto3) API Reference*.
  - Query
  - Scan

For a complete list of AWS SDK developer guides and code examples, see Using DynamoDB with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Conditionally update a DynamoDB item with a TTL using an AWS SDK

The following code examples show how to conditionally update an item's TTL.

Java

SDK for Java 2.x

Update TTL on on an existing DynamoDB Item in a table, with a condition.

```
package com.amazon.samplelib.ttl;

import com.amazon.samplelib.CodeSampleUtils;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import
 software.amazon.awssdk.services.dynamodb.model.ConditionalCheckFailedException;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.ResourceNotFoundException;
import software.amazon.awssdk.services.dynamodb.model.UpdateItemRequest;
import software.amazon.awssdk.services.dynamodb.model.UpdateItemResponse;

import java.util.Map;
import java.util.Optional;

/**
 * Updates an item in a DynamoDB table with TTL attributes using a conditional
 expression.
 * This class demonstrates how to conditionally update TTL expiration timestamps.
 */
public class UpdateTTLConditional {

    private static final String USAGE =
        """
            Usage:
                <tableName> <primaryKey> <sortKey> <region>
            Where:
                tableName - The Amazon DynamoDB table being queried.
                primaryKey - The name of the primary key. Also known as the hash
 or partition key.
```

```
                sortKey - The name of the sort key. Also known as the range
attribute.
                region (optional) - The AWS region that the Amazon DynamoDB table
is located in. (Default: us-east-1)
            """;
    private static final int DAYS_TO_EXPIRE = 90;
    private static final int SECONDS_PER_DAY = 24 * 60 * 60;
    private static final String PRIMARY_KEY_ATTR = "primaryKey";
    private static final String SORT_KEY_ATTR = "sortKey";
    private static final String UPDATED_AT_ATTR = "updatedAt";
    private static final String EXPIRE_AT_ATTR = "expireAt";
    private static final String UPDATE_EXPRESSION = "SET " + UPDATED_AT_ATTR +
"=:c, " + EXPIRE_AT_ATTR + "=:e";
    private static final String CONDITION_EXPRESSION = "attribute_exists(" +
PRIMARY_KEY_ATTR + ")";
    private static final String SUCCESS_MESSAGE = "%s UpdateItem operation with
TTL successful.";
    private static final String CONDITION_FAILED_MESSAGE = "Condition check
failed. Item does not exist.";
    private static final String TABLE_NOT_FOUND_ERROR = "Error: The Amazon
DynamoDB table \"%s\" can't be found.";

    private final DynamoDbClient dynamoDbClient;

    /**
     * Constructs an UpdateTTLConditional with a default DynamoDB client.
     */
    public UpdateTTLConditional() {
        this.dynamoDbClient = null;
    }

    /**
     * Constructs an UpdateTTLConditional with the specified DynamoDB client.
     *
     * @param dynamoDbClient The DynamoDB client to use
     */
    public UpdateTTLConditional(final DynamoDbClient dynamoDbClient) {
        this.dynamoDbClient = dynamoDbClient;
    }

    /**
     * Main method to demonstrate conditionally updating an item with TTL.
     *
     * @param args Command line arguments
```

```
     */
    public static void main(final String[] args) {
        try {
            int result = new UpdateTTLConditional().processArgs(args);
            System.exit(result);
        } catch (Exception e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }

    /**
     * Process command line arguments and conditionally update an item with TTL.
     *
     * @param args Command line arguments
     * @return 0 if successful, non-zero otherwise
     * @throws ResourceNotFoundException If the table doesn't exist
     * @throws DynamoDbException If an error occurs during the operation
     * @throws IllegalArgumentException If arguments are invalid
     */
    public int processArgs(final String[] args) {
        // Argument validation (remove or replace this line when reusing this
code)
        CodeSampleUtils.validateArgs(args, new int[] {3, 4}, USAGE);

        final String tableName = args[0];
        final String primaryKey = args[1];
        final String sortKey = args[2];
        final Region region = Optional.ofNullable(args.length > 3 ? args[3] :
null)
            .map(Region::of)
            .orElse(Region.US_EAST_1);

        // Get current time in epoch second format
        final long currentTime = System.currentTimeMillis() / 1000;

        // Calculate expiration time 90 days from now in epoch second format
        final long expireDate = currentTime + (DAYS_TO_EXPIRE * SECONDS_PER_DAY);

        // Create the key map for the item to update
        final Map<String, AttributeValue> keyMap = Map.of(
            PRIMARY_KEY_ATTR, AttributeValue.builder().s(primaryKey).build(),
            SORT_KEY_ATTR, AttributeValue.builder().s(sortKey).build());
```

```
        // Create the expression attribute values
        final Map<String, AttributeValue> expressionAttributeValues = Map.of(
            ":c",
 AttributeValue.builder().n(String.valueOf(currentTime)).build(),
            ":e",
 AttributeValue.builder().n(String.valueOf(expireDate)).build());

        final UpdateItemRequest request = UpdateItemRequest.builder()
            .tableName(tableName)
            .key(keyMap)
            .updateExpression(UPDATE_EXPRESSION)
            .conditionExpression(CONDITION_EXPRESSION)
            .expressionAttributeValues(expressionAttributeValues)
            .build();

        try (DynamoDbClient ddb = dynamoDbClient != null
            ? dynamoDbClient
            : DynamoDbClient.builder().region(region).build()) {
            final UpdateItemResponse response = ddb.updateItem(request);
            System.out.println(String.format(SUCCESS_MESSAGE, tableName));
            return 0;
        } catch (ConditionalCheckFailedException e) {
            System.err.println(CONDITION_FAILED_MESSAGE);
            throw e;
        } catch (ResourceNotFoundException e) {
            System.err.format(TABLE_NOT_FOUND_ERROR, tableName);
            throw e;
        } catch (DynamoDbException e) {
            System.err.println(e.getMessage());
            throw e;
        }
    }
}
```

- For API details, see UpdateItem in *AWS SDK for Java 2.x API Reference*.

JavaScript

SDK for JavaScript (v3)

Update TTL on on an existing DynamoDB Item in a table, with a condition.

```
import { DynamoDBClient, UpdateItemCommand } from "@aws-sdk/client-dynamodb";
import { marshall, unmarshall } from "@aws-sdk/util-dynamodb";

export const updateItemConditional = async (tableName, partitionKey, sortKey,
 region = 'us-east-1', newAttribute = 'default-value') => {
    const client = new DynamoDBClient({
        region: region,
        endpoint: `https://dynamodb.${region}.amazonaws.com`
    });

    const currentTime = Math.floor(Date.now() / 1000);

    const params = {
        TableName: tableName,
        Key: marshall({
            artist: partitionKey,
            album: sortKey
        }),
        UpdateExpression: "SET newAttribute = :newAttribute",
        ConditionExpression: "expireAt > :expiration",
        ExpressionAttributeValues: marshall({
            ':newAttribute': newAttribute,
            ':expiration': currentTime
        }),
        ReturnValues: "ALL_NEW"
    };

    try {
        const response = await client.send(new UpdateItemCommand(params));
        const responseData = unmarshall(response.Attributes);
        console.log("Item updated successfully: ", responseData);
        return responseData;
    } catch (error) {
        if (error.name === "ConditionalCheckFailedException") {
            console.log("Condition check failed: Item's 'expireAt' is expired.");
        } else {
            console.error("Error updating item: ", error);
        }
        throw error;
    }
};

// Example usage (commented out for testing)
```

```
// updateItemConditional('your-table-name', 'your-partition-key-value', 'your-
sort-key-value');
```

- For API details, see UpdateItem in *AWS SDK for JavaScript API Reference*.

Python

SDK for Python (Boto3)

Update TTL on on an existing DynamoDB Item in a table, with a condition.

```
from datetime import datetime, timedelta

import boto3
from botocore.exceptions import ClientError


def update_dynamodb_item_ttl(table_name, region, primary_key, sort_key,
 ttl_attribute):
    """
    Updates an existing record in a DynamoDB table with a new or updated TTL
 attribute.

    :param table_name: Name of the DynamoDB table
    :param region: AWS Region of the table - example `us-east-1`
    :param primary_key: one attribute known as the partition key.
    :param sort_key: Also known as a range attribute.
    :param ttl_attribute: name of the TTL attribute in the target DynamoDB table
    :return:
    """
    try:
        dynamodb = boto3.resource("dynamodb", region_name=region)
        table = dynamodb.Table(table_name)

        # Generate updated TTL in epoch second format
        updated_expiration_time = int((datetime.now() +
 timedelta(days=90)).timestamp())

        # Define the update expression for adding/updating a new attribute
        update_expression = "SET newAttribute = :val1"
```

```
        # Define the condition expression for checking if 'expireAt' is not
 expired
        condition_expression = "expireAt > :val2"

        # Define the expression attribute values
        expression_attribute_values = {":val1": ttl_attribute, ":val2":
 updated_expiration_time}

        response = table.update_item(
            Key={"primaryKey": primary_key, "sortKey": sort_key},
            UpdateExpression=update_expression,
            ConditionExpression=condition_expression,
            ExpressionAttributeValues=expression_attribute_values,
        )

        print("Item updated successfully.")
        return response["ResponseMetadata"]["HTTPStatusCode"]  # Ideally a 200 OK
    except ClientError as e:
        if e.response["Error"]["Code"] == "ConditionalCheckFailedException":
            print("Condition check failed: Item's 'expireAt' is expired.")
        else:
            print(f"Error updating item: {e}")
    except Exception as e:
        print(f"Error updating item: {e}")


# replace with your values
update_dynamodb_item_ttl(
    "your-table-name",
    "us-east-1",
    "your-partition-key-value",
    "your-sort-key-value",
    "your-ttl-attribute-value",
)
```

- For API details, see UpdateItem in *AWS SDK for Python (Boto3) API Reference*.

For a complete list of AWS SDK developer guides and code examples, see Using DynamoDB with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Connect to a local DynamoDB instance using an AWS SDK

The following code example shows how to override an endpoint URL to connect to a local development deployment of DynamoDB and an AWS SDK.

For more information, see DynamoDB Local.

Rust

SDK for Rust

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
/// Lists your tables from a local DynamoDB instance by setting the SDK Config's
/// endpoint_url and test_credentials.
#[tokio::main]
async fn main() {
    tracing_subscriber::fmt::init();

    let config = aws_config::defaults(aws_config::BehaviorVersion::latest())
        .test_credentials()
        // DynamoDB run locally uses port 8000 by default.
        .endpoint_url("http://localhost:8000")
        .load()
        .await;
    let dynamodb_local_config =
 aws_sdk_dynamodb::config::Builder::from(&config).build();

    let client = aws_sdk_dynamodb::Client::from_conf(dynamodb_local_config);

    let list_resp = client.list_tables().send().await;
    match list_resp {
        Ok(resp) => {
            println!("Found {} tables", resp.table_names().len());
            for name in resp.table_names() {
                println!("  {}", name);
            }
```

```
        }
        Err(err) => eprintln!("Failed to list local dynamodb tables: {err:?}"),
    }
}
```

For a complete list of AWS SDK developer guides and code examples, see Using DynamoDB with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Count expression operators in DynamoDB with an AWS SDK

The following code examples show how to count expression operators in DynamoDB.

- Understand DynamoDB's 300 operator limit.
- Count operators in complex expressions.
- Optimize expressions to stay within limits.

Java

SDK for Java 2.x

Demonstrate expression operator counting using AWS SDK for Java 2.x.

```
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.UpdateItemRequest;
import software.amazon.awssdk.services.dynamodb.model.UpdateItemResponse;

import java.util.HashMap;
import java.util.Map;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

    /**
     * Creates a complex filter expression with a specified number of conditions.
     *
     * <p>This method demonstrates how to generate a complex expression with
     * a specific number of operators to test the 300 operator limit.
```

```
    *
    * @param conditionsCount Number of conditions to include
    * @param useAnd Whether to use AND (true) or OR (false) between conditions
    * @return Map containing the filter expression, attribute values, and
operator count
    */
   public static Map<String, Object> createComplexFilterExpression(int
conditionsCount, boolean useAnd) {
       // Initialize the expression parts and attribute values
       StringBuilder filterExpression = new StringBuilder();
       Map<String, AttributeValue> expressionAttributeValues = new HashMap<>();

       // Generate the specified number of conditions
       for (int i = 0; i < conditionsCount; i++) {
           // Add the operator between conditions (except for the first one)
           if (i > 0) {
               filterExpression.append(useAnd ? " AND " : " OR ");
           }

           // Alternate between different comparison operators for variety
           String valueKey = ":val" + i;

           switch (i % 5) {
               case 0:
                   filterExpression.append("attribute").append(i).append(" =
").append(valueKey);
                   expressionAttributeValues.put(
                       valueKey, AttributeValue.builder().s("value" +
i).build());
                   break;
               case 1:
                   filterExpression.append("attribute").append(i).append(" >
").append(valueKey);
                   expressionAttributeValues.put(
                       valueKey,
AttributeValue.builder().n(String.valueOf(i)).build());
                   break;
               case 2:
                   filterExpression.append("attribute").append(i).append(" <
").append(valueKey);
                   expressionAttributeValues.put(
                       valueKey,
                       AttributeValue.builder().n(String.valueOf(i *
10)).build());
```

```
                    break;
                case 3:
                    filterExpression
                        .append("contains(attribute")
                        .append(i)
                        .append(", ")
                        .append(valueKey)
                        .append(")");
                    expressionAttributeValues.put(
                        valueKey, AttributeValue.builder().s("substring" +
i).build());
                    break;
                case 4:
                    filterExpression
                        .append("attribute_exists(attribute")
                        .append(i)
                        .append(")");
                    break;
                default:
                    // This case will never be reached, but added to satisfy
checkstyle
                    break;
            }
        }

        // Calculate the operator count
        // Each condition has 1 operator (=, >, <, contains, attribute_exists)
        // Each AND or OR between conditions is 1 operator
        int operatorCount = conditionsCount + (conditionsCount > 0 ?
conditionsCount - 1 : 0);

        // Create the result map
        Map<String, Object> result = new HashMap<>();
        result.put("filterExpression", filterExpression.toString());
        result.put("expressionAttributeValues", expressionAttributeValues);
        result.put("operatorCount", operatorCount);

        return result;
    }

    /**
     * Creates a complex update expression with a specified number of operations.
     *
```

```
    * <p>This method demonstrates how to generate a complex update expression
with
    * a specific number of operators to test the 300 operator limit.
    *
    * @param operationsCount Number of operations to include
    * @return Map containing the update expression, attribute values, and
operator count
    */
   public static Map<String, Object> createComplexUpdateExpression(int
operationsCount) {
       // Initialize the expression parts and attribute values
       StringBuilder updateExpression = new StringBuilder("SET ");
       Map<String, AttributeValue> expressionAttributeValues = new HashMap<>();

       // Generate the specified number of SET operations
       for (int i = 0; i < operationsCount; i++) {
           // Add comma between operations (except for the first one)
           if (i > 0) {
               updateExpression.append(", ");
           }

           // Alternate between different types of SET operations
           String valueKey = ":val" + i;

           switch (i % 3) {
               case 0:
                   // Simple assignment (1 operator: =)
                   updateExpression.append("attribute").append(i).append(" =
").append(valueKey);
                   expressionAttributeValues.put(
                       valueKey, AttributeValue.builder().s("value" +
i).build());
                   break;
               case 1:
                   // Addition (2 operators: = and +)
                   updateExpression
                       .append("attribute")
                       .append(i)
                       .append(" = attribute")
                       .append(i)
                       .append(" + ")
                       .append(valueKey);
                   expressionAttributeValues.put(
```

```
                        valueKey,
AttributeValue.builder().n(String.valueOf(i)).build());
                    break;
                case 2:
                    // Conditional assignment with if_not_exists (2 operators: =
and if_not_exists)
                    updateExpression
                        .append("attribute")
                        .append(i)
                        .append(" = if_not_exists(attribute")
                        .append(i)
                        .append(", ")
                        .append(valueKey)
                        .append(")");
                    expressionAttributeValues.put(
                        valueKey,
                        AttributeValue.builder().n(String.valueOf(i *
10)).build());
                    break;
                default:
                    // This case will never be reached, but added to satisfy
checkstyle
                    break;
            }
        }

        // Calculate the operator count
        // Each operation has 1-2 operators as noted above
        int operatorCount = 0;
        for (int i = 0; i < operationsCount; i++) {
            operatorCount += (i % 3 == 0) ? 1 : 2;
        }

        // Create the result map
        Map<String, Object> result = new HashMap<>();
        result.put("updateExpression", updateExpression.toString());
        result.put("expressionAttributeValues", expressionAttributeValues);
        result.put("operatorCount", operatorCount);

        return result;
    }

    /**
```

```
    * Test the operator limit by attempting an operation with a complex
expression.
    *
    * <p>This method demonstrates what happens when an expression approaches or
    * exceeds the 300 operator limit.
    *
    * @param dynamoDbClient The DynamoDB client
    * @param tableName The name of the DynamoDB table
    * @param key The key of the item to update
    * @param operatorCount Target number of operators to include
    * @return Map containing the result of the operation attempt
    */
   public static Map<String, Object> testOperatorLimit(
       DynamoDbClient dynamoDbClient, String tableName, Map<String,
AttributeValue> key, int operatorCount) {

       // Create a complex update expression with the specified operator count
       Map<String, Object> expressionData =
           createComplexUpdateExpression((int) Math.ceil(operatorCount /
1.5)); // Adjust to get close to target count

       String updateExpression = (String)
expressionData.get("updateExpression");
       @SuppressWarnings("unchecked")
       Map<String, AttributeValue> expressionAttributeValues =
           (Map<String, AttributeValue>)
expressionData.get("expressionAttributeValues");
       int actualCount = (int) expressionData.get("operatorCount");

       System.out.println("Generated update expression with approximately " +
actualCount + " operators");

       // Define the update parameters
       UpdateItemRequest request = UpdateItemRequest.builder()
           .tableName(tableName)
           .key(key)
           .updateExpression(updateExpression)
           .expressionAttributeValues(expressionAttributeValues)
           .returnValues("UPDATED_NEW")
           .build();

       try {
           // Attempt the update operation
           UpdateItemResponse response = dynamoDbClient.updateItem(request);
```

```
            Map<String, Object> result = new HashMap<>();
            result.put("success", true);
            result.put("message", "Operation succeeded with " + actualCount + "
operators");
            result.put("data", response);
            return result;

        } catch (DynamoDbException e) {
            // Check if the error is due to exceeding the operator limit
            if (e.getMessage().contains("too many operators")) {
                Map<String, Object> result = new HashMap<>();
                result.put("success", false);
                result.put("message", "Operation failed: " + e.getMessage());
                result.put("operatorCount", actualCount);
                return result;
            }

            // Return other errors
            Map<String, Object> result = new HashMap<>();
            result.put("success", false);
            result.put("message", "Operation failed: " + e.getMessage());
            result.put("error", e);
            return result;
        }
    }

    /**
     * Break down a complex expression into multiple simpler operations.
     *
     * <p>This method demonstrates how to handle expressions that would exceed
     * the 300 operator limit by breaking them into multiple operations.
     *
     * @param dynamoDbClient The DynamoDB client
     * @param tableName The name of the DynamoDB table
     * @param key The key of the item to update
     * @param totalOperations Total number of operations to perform
     * @return Map containing the results of the operations
     */
    public static Map<String, Object> breakDownComplexExpression(
        DynamoDbClient dynamoDbClient, String tableName, Map<String,
AttributeValue> key, int totalOperations) {

        // Calculate how many operations we can safely include in each batch
```

```
        // Using 150 as a conservative limit (well below 300)
        final int operationsPerBatch = 100;
        final int batchCount = (int) Math.ceil((double) totalOperations /
operationsPerBatch);

        System.out.println("Breaking down " + totalOperations + " operations into
" + batchCount + " batches");

        Map<String, Object> results = new HashMap<>();
        results.put("totalBatches", batchCount);

        Map<Integer, Map<String, Object>> batchResults = new HashMap<>();

        // Process each batch
        for (int batch = 0; batch < batchCount; batch++) {
            // Calculate the operations for this batch
            int batchStart = batch * operationsPerBatch;
            int batchEnd = Math.min(batchStart + operationsPerBatch,
totalOperations);
            int batchSize = batchEnd - batchStart;

            System.out.println(
                "Processing batch " + (batch + 1) + "/" + batchCount + " with " +
batchSize + " operations");

            // Create an update expression for this batch
            Map<String, Object> expressionData =
createComplexUpdateExpression(batchSize);

            String updateExpression = (String)
expressionData.get("updateExpression");
            @SuppressWarnings("unchecked")
            Map<String, AttributeValue> expressionAttributeValues =
                (Map<String, AttributeValue>)
expressionData.get("expressionAttributeValues");
            int operatorCount = (int) expressionData.get("operatorCount");

            // Define the update parameters
            UpdateItemRequest request = UpdateItemRequest.builder()
                .tableName(tableName)
                .key(key)
                .updateExpression(updateExpression)
                .expressionAttributeValues(expressionAttributeValues)
                .returnValues("UPDATED_NEW")
```

```
                .build();

            try {
                // Perform the update operation for this batch
                UpdateItemResponse response = dynamoDbClient.updateItem(request);

                Map<String, Object> batchResult = new HashMap<>();
                batchResult.put("batch", batch + 1);
                batchResult.put("success", true);
                batchResult.put("operatorCount", operatorCount);
                batchResult.put("attributes", response.attributes());

                batchResults.put(batch, batchResult);

            } catch (DynamoDbException e) {
                Map<String, Object> batchResult = new HashMap<>();
                batchResult.put("batch", batch + 1);
                batchResult.put("success", false);
                batchResult.put("operatorCount", operatorCount);
                batchResult.put("error", e.getMessage());

                batchResults.put(batch, batchResult);

                // Continue with next batch instead of breaking
                continue;
            }
        }

        results.put("results", batchResults);
        return results;
    }

    /**
     * Count operators in a DynamoDB expression based on the rules in the
documentation.
     *
     * <p>This method demonstrates how operators are counted according to the
     * DynamoDB documentation.
     *
     * @param expression The DynamoDB expression to analyze
     * @return Map containing the breakdown of operator counts
     */
    public static Map<String, Integer> countOperatorsInExpression(String
expression) {
```

```
        // Initialize counters for different operator types
        Map<String, Integer> counts = new HashMap<>();
        counts.put("comparisonOperators", 0);
        counts.put("logicalOperators", 0);
        counts.put("functions", 0);
        counts.put("arithmeticOperators", 0);
        counts.put("specialOperators", 0);
        counts.put("total", 0);

        // Count comparison operators (=, <>, <, <=, >, >=)
        // This is a simplified approach and may not catch all cases
        int comparisonCount = 0;
        Pattern comparisonPattern = Pattern.compile("(=|<>|<=|>=|<|>)");
        Matcher comparisonMatcher = comparisonPattern.matcher(expression);
        while (comparisonMatcher.find()) {
            comparisonCount++;
        }
        counts.put("comparisonOperators", comparisonCount);

        // Count logical operators (AND, OR, NOT)
        int andCount = countOccurrences(expression, "\\bAND\\b");
        int orCount = countOccurrences(expression, "\\bOR\\b");
        int notCount = countOccurrences(expression, "\\bNOT\\b");
        counts.put("logicalOperators", andCount + orCount + notCount);

        // Count functions (attribute_exists, attribute_not_exists,
 attribute_type, begins_with, contains, size)
        int functionCount = countOccurrences(
            expression,
            "\\b(attribute_exists|attribute_not_exists|attribute_type|
begins_with|contains|size|if_not_exists)\\(");
        counts.put("functions", functionCount);

        // Count arithmetic operators (+ and -)
        // This is a simplified approach and may not catch all cases
        int arithmeticCount = 0;
        Pattern arithmeticPattern = Pattern.compile("[a-zA-Z0-9_)\\]]\\s*[\\+\
\-]\\s*[a-zA-Z0-9_:(]");
        Matcher arithmeticMatcher = arithmeticPattern.matcher(expression);
        while (arithmeticMatcher.find()) {
            arithmeticCount++;
        }
        counts.put("arithmeticOperators", arithmeticCount);
```

```
        // Count special operators (BETWEEN, IN)
        int betweenCount = countOccurrences(expression, "\\bBETWEEN\\b");
        int inCount = countOccurrences(expression, "\\bIN\\b");
        counts.put("specialOperators", betweenCount + inCount);

        // Add extra operators for BETWEEN (each BETWEEN includes an AND)
        int currentLogicalOps = counts.getOrDefault("logicalOperators", 0);
        counts.put("logicalOperators", currentLogicalOps + betweenCount);

        // Calculate total
        int total = counts.getOrDefault("comparisonOperators", 0)
            + counts.getOrDefault("logicalOperators", 0)
            + counts.getOrDefault("functions", 0)
            + counts.getOrDefault("arithmeticOperators", 0)
            + counts.getOrDefault("specialOperators", 0);
        counts.put("total", total);

        return counts;
    }

    /**
     * Helper method to count occurrences of a pattern in a string.
     *
     * @param text The text to search in
     * @param regex The regular expression pattern to search for
     * @return The number of occurrences
     */
    private static int countOccurrences(String text, String regex) {
        final Pattern pattern = Pattern.compile(regex);
        final Matcher matcher = pattern.matcher(text);
        int count = 0;
        while (matcher.find()) {
            count++;
        }
        return count;
    }
```

Example usage of expression operator counting with AWS SDK for Java 2.x.

```
    public static void exampleUsage(DynamoDbClient dynamoDbClient, String
tableName) {
        // Example key
```

```
        Map<String, AttributeValue> key = new HashMap<>();
        key.put("ProductId", AttributeValue.builder().s("P12345").build());

        System.out.println("Demonstrating DynamoDB expression operator counting
and the 300 operator limit");

        try {
            // Example 1: Analyze a simple expression
            System.out.println("\nExample 1: Analyzing a simple expression");
            String simpleExpression = "Price = :price AND Rating > :rating AND
Category IN (:cat1, :cat2, :cat3)";
            Map<String, Integer> simpleCount =
countOperatorsInExpression(simpleExpression);

            System.out.println("Expression: " + simpleExpression);
            System.out.println("Operator count breakdown:");
            System.out.println("- Comparison operators: " +
simpleCount.get("comparisonOperators"));
            System.out.println("- Logical operators: " +
simpleCount.get("logicalOperators"));
            System.out.println("- Functions: " + simpleCount.get("functions"));
            System.out.println("- Arithmetic operators: " +
simpleCount.get("arithmeticOperators"));
            System.out.println("- Special operators: " +
simpleCount.get("specialOperators"));
            System.out.println("- Total operators: " + simpleCount.get("total"));

            // Example 2: Analyze a complex expression
            System.out.println("\nExample 2: Analyzing a complex expression");
            String complexExpression = "(attribute_exists(Category) AND Size
BETWEEN :min AND :max) OR "
                + "(Price > :price AND contains(Description, :keyword) AND "
                + "(Rating >= :minRating OR Reviews > :minReviews))";
            Map<String, Integer> complexCount =
countOperatorsInExpression(complexExpression);

            System.out.println("Expression: " + complexExpression);
            System.out.println("Operator count breakdown:");
            System.out.println("- Comparison operators: " +
complexCount.get("comparisonOperators"));
            System.out.println("- Logical operators: " +
complexCount.get("logicalOperators"));
            System.out.println("- Functions: " + complexCount.get("functions"));
```

```
            System.out.println("- Arithmetic operators: " +
complexCount.get("arithmeticOperators"));
            System.out.println("- Special operators: " +
complexCount.get("specialOperators"));
            System.out.println("- Total operators: " +
complexCount.get("total"));

            // Example 3: Test approaching the operator limit
            System.out.println("\nExample 3: Testing an expression approaching
the operator limit");
            Map<String, Object> approachingLimit =
testOperatorLimit(dynamoDbClient, tableName, key, 290);
            System.out.println(approachingLimit.get("message"));

            // Example 4: Test exceeding the operator limit
            System.out.println("\nExample 4: Testing an expression exceeding the
operator limit");
            Map<String, Object> exceedingLimit =
testOperatorLimit(dynamoDbClient, tableName, key, 310);
            System.out.println(exceedingLimit.get("message"));

            // Example 5: Breaking down a complex expression
            System.out.println("\nExample 5: Breaking down a complex expression
into multiple operations");
            Map<String, Object> breakdownResult =
breakDownComplexExpression(dynamoDbClient, tableName, key, 500);
            @SuppressWarnings("unchecked")
            Map<Integer, Map<String, Object>> results =
                (Map<Integer, Map<String, Object>>)
breakdownResult.get("results");
            System.out.println(
                "Processed " + results.size() + " of " +
breakdownResult.get("totalBatches") + " batches");

            // Explain the operator counting rules
            System.out.println("\nKey points about DynamoDB expression operator
counting:");
            System.out.println("1. The maximum number of operators in any
expression is 300");
            System.out.println("2. Each comparison operator (=, <>, <, <=, >, >=)
counts as 1 operator");
            System.out.println("3. Each logical operator (AND, OR, NOT) counts as
1 operator");
```

```
            System.out.println("4. Each function call (attribute_exists,
 contains, etc.) counts as 1 operator");
            System.out.println("5. Each arithmetic operator (+ or -) counts as 1
 operator");
            System.out.println("6. BETWEEN counts as 2 operators (BETWEEN itself
 and the AND within it)");
            System.out.println("7. IN counts as 1 operator regardless of the
 number of values");
            System.out.println("8. Parentheses for grouping and attribute paths
 don't count as operators");
            System.out.println("9. When you exceed the limit, the error always
 reports '301 operators'");
            System.out.println("10. For complex operations, break them into
 multiple smaller operations");

        } catch (Exception e) {
            System.err.println("Error: " + e.getMessage());
            e.printStackTrace();
        }
    }
```

- For API details, see [UpdateItem](#) in *AWS SDK for Java 2.x API Reference*.

JavaScript

SDK for JavaScript (v3)

Demonstrate expression operator counting using AWS SDK for JavaScript.

```
const { DynamoDBClient } = require("@aws-sdk/client-dynamodb");
const {
  DynamoDBDocumentClient,
  UpdateCommand,
  QueryCommand
} = require("@aws-sdk/lib-dynamodb");

/**
 * Create a complex filter expression with a specified number of conditions.
 *
 * This function demonstrates how to generate a complex expression with
 * a specific number of operators to test the 300 operator limit.
 *
```

```
 * @param {number} conditionsCount - Number of conditions to include
 * @param {boolean} useAnd - Whether to use AND (true) or OR (false) between
 conditions
 * @returns {Object} - Object containing the filter expression and attribute
 values
 */
function createComplexFilterExpression(conditionsCount, useAnd = true) {
  // Initialize the expression parts and attribute values
  const conditions = [];
  const expressionAttributeValues = {};

  // Generate the specified number of conditions
  for (let i = 0; i < conditionsCount; i++) {
    // Alternate between different comparison operators for variety
    let condition;
    const valueKey = `:val${i}`;

    switch (i % 5) {
      case 0:
        condition = `attribute${i} = ${valueKey}`;
        expressionAttributeValues[valueKey] = `value${i}`;
        break;
      case 1:
        condition = `attribute${i} > ${valueKey}`;
        expressionAttributeValues[valueKey] = i;
        break;
      case 2:
        condition = `attribute${i} < ${valueKey}`;
        expressionAttributeValues[valueKey] = i * 10;
        break;
      case 3:
        condition = `contains(attribute${i}, ${valueKey})`;
        expressionAttributeValues[valueKey] = `substring${i}`;
        break;
      case 4:
        condition = `attribute_exists(attribute${i})`;
        break;
    }

    conditions.push(condition);
  }

  // Join the conditions with AND or OR
  const operator = useAnd ? " AND " : " OR ";
```

```
  const filterExpression = conditions.join(operator);

  // Calculate the operator count
  // Each condition has 1 operator (=, >, <, contains, attribute_exists)
  // Each AND or OR between conditions is 1 operator
  const operatorCount = conditionsCount + (conditionsCount > 0 ? conditionsCount
 - 1 : 0);

  return {
    filterExpression,
    expressionAttributeValues,
    operatorCount
  };
}

/**
 * Create a complex update expression with a specified number of operations.
 *
 * This function demonstrates how to generate a complex update expression with
 * a specific number of operators to test the 300 operator limit.
 *
 * @param {number} operationsCount - Number of operations to include
 * @returns {Object} - Object containing the update expression and attribute
 values
 */
function createComplexUpdateExpression(operationsCount) {
  // Initialize the expression parts and attribute values
  const setOperations = [];
  const expressionAttributeValues = {};

  // Generate the specified number of SET operations
  for (let i = 0; i < operationsCount; i++) {
    // Alternate between different types of SET operations
    let operation;
    const valueKey = `:val${i}`;

    switch (i % 3) {
      case 0:
        // Simple assignment (1 operator: =)
        operation = `attribute${i} = ${valueKey}`;
        expressionAttributeValues[valueKey] = `value${i}`;
        break;
      case 1:
        // Addition (2 operators: = and +)
```

```
        operation = `attribute${i} = attribute${i} + ${valueKey}`;
        expressionAttributeValues[valueKey] = i;
        break;
      case 2:
        // Conditional assignment with if_not_exists (2 operators: = and
 if_not_exists)
        operation = `attribute${i} = if_not_exists(attribute${i}, ${valueKey})`;
        expressionAttributeValues[valueKey] = i * 10;
        break;
    }

    setOperations.push(operation);
  }

  // Create the update expression
  const updateExpression = `SET ${setOperations.join(", ")}`;

  // Calculate the operator count
  // Each operation has 1-2 operators as noted above
  let operatorCount = 0;
  for (let i = 0; i < operationsCount; i++) {
    operatorCount += (i % 3 === 0) ? 1 : 2;
  }

  return {
    updateExpression,
    expressionAttributeValues,
    operatorCount
  };
}

/**
 * Test the operator limit by attempting an operation with a complex expression.
 *
 * This function demonstrates what happens when an expression approaches or
 * exceeds the 300 operator limit.
 *
 * @param {Object} config - AWS configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {Object} key - The key of the item to update
 * @param {number} operatorCount - Target number of operators to include
 * @returns {Promise<Object>} - Result of the operation attempt
 */
async function testOperatorLimit(
```

```
  config,
  tableName,
  key,
  operatorCount
) {
  // Initialize the DynamoDB client
  const client = new DynamoDBClient(config);
  const docClient = DynamoDBDocumentClient.from(client);

  // Create a complex update expression with the specified operator count
  const { updateExpression, expressionAttributeValues, operatorCount:
 actualCount } =
    createComplexUpdateExpression(Math.ceil(operatorCount / 1.5)); // Adjust to
 get close to target count

  console.log(`Generated update expression with approximately ${actualCount}
 operators`);

  // Define the update parameters
  const params = {
    TableName: tableName,
    Key: key,
    UpdateExpression: updateExpression,
    ExpressionAttributeValues: expressionAttributeValues,
    ReturnValues: "UPDATED_NEW"
  };

  try {
    // Attempt the update operation
    const response = await docClient.send(new UpdateCommand(params));
    return {
      success: true,
      message: `Operation succeeded with ${actualCount} operators`,
      data: response
    };
  } catch (error) {
    // Check if the error is due to exceeding the operator limit
    if (error.name === "ValidationException" &&
        error.message.includes("too many operators")) {
      return {
        success: false,
        message: `Operation failed: ${error.message}`,
        operatorCount: actualCount
      };
```

```
    }

    // Return other errors
    return {
      success: false,
      message: `Operation failed: ${error.message}`,
      error
    };
  }
}

/**
 * Break down a complex expression into multiple simpler operations.
 *
 * This function demonstrates how to handle expressions that would exceed
 * the 300 operator limit by breaking them into multiple operations.
 *
 * @param {Object} config - AWS configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {Object} key - The key of the item to update
 * @param {number} totalOperations - Total number of operations to perform
 * @returns {Promise<Object>} - Result of the operations
 */
async function breakDownComplexExpression(
  config,
  tableName,
  key,
  totalOperations
) {
  // Initialize the DynamoDB client
  const client = new DynamoDBClient(config);
  const docClient = DynamoDBDocumentClient.from(client);

  // Calculate how many operations we can safely include in each batch
  // Using 150 as a conservative limit (well below 300)
  const operationsPerBatch = 100;
  const batchCount = Math.ceil(totalOperations / operationsPerBatch);

  console.log(`Breaking down ${totalOperations} operations into ${batchCount}
 batches`);

  const results = [];

  // Process each batch
```

```
 for (let batch = 0; batch < batchCount; batch++) {
   // Calculate the operations for this batch
   const batchStart = batch * operationsPerBatch;
   const batchEnd = Math.min(batchStart + operationsPerBatch, totalOperations);
   const batchSize = batchEnd - batchStart;

   console.log(`Processing batch ${batch + 1}/${batchCount} with ${batchSize}
operations`);

   // Create an update expression for this batch
   const { updateExpression, expressionAttributeValues, operatorCount } =
     createComplexUpdateExpression(batchSize);

   // Define the update parameters
   const params = {
     TableName: tableName,
     Key: key,
     UpdateExpression: updateExpression,
     ExpressionAttributeValues: expressionAttributeValues,
     ReturnValues: "UPDATED_NEW"
   };

   try {
     // Perform the update operation for this batch
     const response = await docClient.send(new UpdateCommand(params));

     results.push({
       batch: batch + 1,
       success: true,
       operatorCount,
       attributes: response.Attributes
     });
   } catch (error) {
     results.push({
       batch: batch + 1,
       success: false,
       operatorCount,
       error: error.message
     });

     // Stop processing if an error occurs
     break;
   }
 }
```

```
  return {
    totalBatches: batchCount,
    results
  };
}

/**
 * Count operators in a DynamoDB expression based on the rules in the
 documentation.
 *
 * This function demonstrates how operators are counted according to the
 * DynamoDB documentation.
 *
 * @param {string} expression - The DynamoDB expression to analyze
 * @returns {Object} - Breakdown of operator counts
 */
function countOperatorsInExpression(expression) {
  // Initialize counters for different operator types
  const counts = {
    comparisonOperators: 0,
    logicalOperators: 0,
    functions: 0,
    arithmeticOperators: 0,
    specialOperators: 0,
    total: 0
  };

  // Count comparison operators (=, <>, <, <=, >, >=)
  const comparisonRegex = /[^<>]=[^=]|<>|<=|>=|[^<]>[^=]|[^>]<[^=]/g;
  const comparisonMatches = expression.match(comparisonRegex) || [];
  counts.comparisonOperators = comparisonMatches.length;

  // Count logical operators (AND, OR, NOT)
  const andMatches = expression.match(/\bAND\b/g) || [];
  const orMatches = expression.match(/\bOR\b/g) || [];
  const notMatches = expression.match(/\bNOT\b/g) || [];
  counts.logicalOperators = andMatches.length + orMatches.length +
 notMatches.length;

  // Count functions (attribute_exists, attribute_not_exists, attribute_type,
 begins_with, contains, size)
  const functionRegex = /\b(attribute_exists|attribute_not_exists|attribute_type|
begins_with|contains|size|if_not_exists)\(/g;
```

```
  const functionMatches = expression.match(functionRegex) || [];
  counts.functions = functionMatches.length;

  // Count arithmetic operators (+ and -)
  const arithmeticMatches = expression.match(/[a-zA-Z0-9_)\]]\s*[\+\-]\s*[a-zA-
Z0-9_(:]/g) || [];
  counts.arithmeticOperators = arithmeticMatches.length;

  // Count special operators (BETWEEN, IN)
  const betweenMatches = expression.match(/\bBETWEEN\b/g) || [];
  const inMatches = expression.match(/\bIN\b/g) || [];
  counts.specialOperators = betweenMatches.length + inMatches.length;

  // Add extra operators for BETWEEN (each BETWEEN includes an AND)
  counts.logicalOperators += betweenMatches.length;

  // Calculate total
  counts.total = counts.comparisonOperators +
                 counts.logicalOperators +
                 counts.functions +
                 counts.arithmeticOperators +
                 counts.specialOperators;

  return counts;
}
```

Example usage of expression operator counting with AWS SDK for JavaScript.

```
/**
 * Example of how to work with expression operator counting.
 */
async function exampleUsage() {
  // Example parameters
  const config = { region: "us-west-2" };
  const tableName = "Products";
  const key = { ProductId: "P12345" };

  console.log("Demonstrating DynamoDB expression operator counting and the 300
 operator limit");

  try {
    // Example 1: Analyze a simple expression
```

```
   console.log("\nExample 1: Analyzing a simple expression");
   const simpleExpression = "Price = :price AND Rating > :rating AND Category IN
(:cat1, :cat2, :cat3)";
   const simpleCount = countOperatorsInExpression(simpleExpression);

   console.log(`Expression: ${simpleExpression}`);
   console.log("Operator count breakdown:");
   console.log(`- Comparison operators: ${simpleCount.comparisonOperators}`);
   console.log(`- Logical operators: ${simpleCount.logicalOperators}`);
   console.log(`- Functions: ${simpleCount.functions}`);
   console.log(`- Arithmetic operators: ${simpleCount.arithmeticOperators}`);
   console.log(`- Special operators: ${simpleCount.specialOperators}`);
   console.log(`- Total operators: ${simpleCount.total}`);

   // Example 2: Analyze a complex expression
   console.log("\nExample 2: Analyzing a complex expression");
   const complexExpression =
     "(attribute_exists(Category) AND Size BETWEEN :min AND :max) OR " +
     "(Price > :price AND contains(Description, :keyword) AND " +
     "(Rating >= :minRating OR Reviews > :minReviews))";
   const complexCount = countOperatorsInExpression(complexExpression);

   console.log(`Expression: ${complexExpression}`);
   console.log("Operator count breakdown:");
   console.log(`- Comparison operators: ${complexCount.comparisonOperators}`);
   console.log(`- Logical operators: ${complexCount.logicalOperators}`);
   console.log(`- Functions: ${complexCount.functions}`);
   console.log(`- Arithmetic operators: ${complexCount.arithmeticOperators}`);
   console.log(`- Special operators: ${complexCount.specialOperators}`);
   console.log(`- Total operators: ${complexCount.total}`);

   // Example 3: Test approaching the operator limit
   console.log("\nExample 3: Testing an expression approaching the operator
limit");
   const approachingLimit = await testOperatorLimit(config, tableName, key,
290);
   console.log(approachingLimit.message);

   // Example 4: Test exceeding the operator limit
   console.log("\nExample 4: Testing an expression exceeding the operator
limit");
   const exceedingLimit = await testOperatorLimit(config, tableName, key, 310);
   console.log(exceedingLimit.message);
```

```
    // Example 5: Breaking down a complex expression
    console.log("\nExample 5: Breaking down a complex expression into multiple
 operations");
    const breakdownResult = await breakDownComplexExpression(config, tableName,
 key, 500);
    console.log(`Processed ${breakdownResult.results.length} of
 ${breakdownResult.totalBatches} batches`);

    // Explain the operator counting rules
    console.log("\nKey points about DynamoDB expression operator counting:");
    console.log("1. The maximum number of operators in any expression is 300");
    console.log("2. Each comparison operator (=, <>, <, <=, >, >=) counts as 1
 operator");
    console.log("3. Each logical operator (AND, OR, NOT) counts as 1 operator");
    console.log("4. Each function call (attribute_exists, contains, etc.) counts
 as 1 operator");
    console.log("5. Each arithmetic operator (+ or -) counts as 1 operator");
    console.log("6. BETWEEN counts as 2 operators (BETWEEN itself and the AND
 within it)");
    console.log("7. IN counts as 1 operator regardless of the number of values");
    console.log("8. Parentheses for grouping and attribute paths don't count as
 operators");
    console.log("9. When you exceed the limit, the error always reports '301
 operators'");
    console.log("10. For complex operations, break them into multiple smaller
 operations");

  } catch (error) {
    console.error("Error:", error);
  }
}
```

- For API details, see UpdateItem in *AWS SDK for JavaScript API Reference*.

Python

SDK for Python (Boto3)

Demonstrate expression operator counting using AWS SDK for Python (Boto3).

```
import boto3
from botocore.exceptions import ClientError
```

```
from typing import Any, Dict, List, Optional, Tuple



def create_complex_filter_expression(
    attribute_name: str, values: List[Any], use_or: bool = True
) -> Tuple[str, Dict[str, Any], Dict[str, str], int]:
    """
    Create a complex filter expression with multiple conditions.

    This function demonstrates how to build a complex filter expression
    and count the number of operators used.

    Args:
        attribute_name (str): The name of the attribute to filter on.
        values (List[Any]): List of values to compare against.
        use_or (bool, optional): Whether to use OR between conditions. Defaults
 to True.

    Returns:
        Tuple[str, Dict[str, Any], Dict[str, str], int]: A tuple containing:
            - The filter expression string
            - Expression attribute values
            - Expression attribute names
            - The number of operators used
    """
    if not values:
        return "", {}, {}, 0

    # Initialize expression components
    filter_expression = ""
    expression_attribute_values = {}
    expression_attribute_names = {"#attr": attribute_name}
    operator_count = 0

    # Build the filter expression
    for i, value in enumerate(values):
        value_placeholder = f":val{i}"
        expression_attribute_values[value_placeholder] = value

        if i > 0:
            # Add OR or AND operator between conditions
            filter_expression += " OR " if use_or else " AND "
            operator_count += 1  # Count the OR/AND operator
```

```
        # Add the condition
        filter_expression += f"#attr = {value_placeholder}"
        operator_count += 1  # Count the = operator

    return (
        filter_expression,
        expression_attribute_values,
        expression_attribute_names,
        operator_count,
    )


def create_nested_filter_expression(
    depth: int, conditions_per_level: int
) -> Tuple[str, Dict[str, Any], Dict[str, str], int]:
    """
    Create a deeply nested filter expression with multiple conditions.

    This function demonstrates how to build a complex nested filter expression
    and count the number of operators used.

    Args:
        depth (int): The depth of nesting.
        conditions_per_level (int): Number of conditions at each level.

    Returns:
        Tuple[str, Dict[str, Any], Dict[str, str], int]: A tuple containing:
            - The filter expression string
            - Expression attribute values
            - Expression attribute names
            - The number of operators used
    """
    if depth <= 0 or conditions_per_level <= 0:
        return "", {}, {}, 0

    # Initialize expression components
    expression_attribute_values = {}
    expression_attribute_names = {}
    operator_count = 0

    def build_nested_expression(current_depth: int, prefix: str) -> str:
        nonlocal operator_count
```

```
        if current_depth <= 0:
            return ""

        # Build conditions at this level
        conditions = []
        for i in range(conditions_per_level):
            attr_name = f"attr{prefix}_{i}"
            attr_placeholder = f"#attr{prefix}_{i}"
            val_placeholder = f":val{prefix}_{i}"

            expression_attribute_names[attr_placeholder] = attr_name
            expression_attribute_values[val_placeholder] = i

            conditions.append(f"{attr_placeholder} = {val_placeholder}")
            operator_count += 1  # Count the = operator

        # Join conditions with AND
        level_expression = " AND ".join(conditions)
        operator_count += max(0, len(conditions) - 1)  # Count the AND operators

        # If not at the deepest level, add nested expressions
        if current_depth > 1:
            nested_expr = build_nested_expression(current_depth - 1,
 f"{prefix}_{current_depth}")
            if nested_expr:
                level_expression = f"({level_expression}) OR ({nested_expr})"
                operator_count += 1  # Count the OR operator

        return level_expression

    # Build the expression starting from the top level
    filter_expression = build_nested_expression(depth, "1")

    return (
        filter_expression,
        expression_attribute_values,
        expression_attribute_names,
        operator_count,
    )


def count_operators_in_update_expression(update_expression: str) -> int:
    """
    Count the number of operators in an update expression.
```

```
    This function demonstrates how to count operators in an update expression
    based on DynamoDB's rules.

    Args:
        update_expression (str): The update expression to analyze.

    Returns:
        int: The number of operators in the expression.
    """
    operator_count = 0

    # Count SET operations
    if "SET" in update_expression:
        set_section = (
            update_expression.split("SET")[1].split("REMOVE")[0].split("ADD")
[0].split("DELETE")[0]
        )

        # Count assignment operators (=)
        operator_count += set_section.count("=")

        # Count arithmetic operators (+, -)
        operator_count += set_section.count("+")
        operator_count += set_section.count("-")

        # Count list_append function calls (each counts as 1 operator)
        operator_count += set_section.lower().count("list_append")

        # Count if_not_exists function calls (each counts as 1 operator)
        operator_count += set_section.lower().count("if_not_exists")

    # Count REMOVE operations (no additional operators)

    # Count ADD operations (each ADD counts as 1 operator)
    if "ADD" in update_expression:
        add_section = (
            update_expression.split("ADD")[1].split("DELETE")[0].split("SET")
[0].split("REMOVE")[0]
        )
        operator_count += add_section.count(",") + 1

    # Count DELETE operations (each DELETE counts as 1 operator)
    if "DELETE" in update_expression:
```

```
        delete_section = (
            update_expression.split("DELETE")[1].split("SET")[0].split("ADD")
[0].split("REMOVE")[0]
        )
        operator_count += delete_section.count(",") + 1

    return operator_count


def count_operators_in_condition_expression(condition_expression: str) -> int:
    """
    Count the number of operators in a condition expression.

    This function demonstrates how to count operators in a condition expression
    based on DynamoDB's rules.

    Args:
        condition_expression (str): The condition expression to analyze.

    Returns:
        int: The number of operators in the expression.
    """
    operator_count = 0

    # Count comparison operators
    comparison_operators = ["=", "<>", "<", "<=", ">", ">="]
    for op in comparison_operators:
        operator_count += condition_expression.count(op)

    # Count logical operators
    operator_count += condition_expression.upper().count(" AND ")
    operator_count += condition_expression.upper().count(" OR ")
    operator_count += condition_expression.upper().count("NOT ")

    # Count BETWEEN operator (counts as 2: BETWEEN + AND)
    between_count = condition_expression.upper().count(" BETWEEN ")
    operator_count += between_count * 2

    # Count IN operator (counts as 1 regardless of number of values)
    operator_count += condition_expression.upper().count(" IN ")

    # Count functions (each counts as 1 operator)
    functions = [
        "attribute_exists",
```

```
        "attribute_not_exists",
        "attribute_type",
        "begins_with",
        "contains",
        "size",
    ]
    for func in functions:
        operator_count += condition_expression.lower().count(func)

    return operator_count


# Note: This function is for demonstration purposes only and should be called
 from example_usage()
# It's not meant to be used directly as a test function
def _test_expression_limit(
    table_name: str, key: Dict[str, Any], operator_count: int, attribute_name:
 str = "TestAttribute"
) -> Tuple[bool, Optional[str]]:
    """
    Test if an expression with a specific number of operators exceeds the limit.

    This function demonstrates how to test the 300 operator limit by creating
    an expression with a specified number of operators.

    Args:
        table_name (str): The name of the DynamoDB table.
        key (Dict[str, Any]): The primary key of the item to update.
        operator_count (int): The number of operators to include in the
 expression.
        attribute_name (str, optional): The name of the attribute to update.
 Defaults to "TestAttribute".

    Returns:
        Tuple[bool, Optional[str]]: A tuple containing:
            - A boolean indicating if the operation succeeded
            - The error message if it failed, None otherwise
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # Create an update expression with the specified number of operators
    update_expression = f"SET #{attribute_name} = :val0"
```

```
    expression_attribute_names = {f"#{attribute_name}": attribute_name}
    expression_attribute_values = {":val0": 0}

    # Add additional SET operations to reach the desired operator count
    # Each assignment adds 1 operator
    for i in range(1, operator_count):
        attr_name = f"{attribute_name}{i}"
        attr_placeholder = f"#attr{i}"
        val_placeholder = f":val{i}"

        update_expression += f", {attr_placeholder} = {val_placeholder}"
        expression_attribute_names[attr_placeholder] = attr_name
        expression_attribute_values[val_placeholder] = i

    try:
        # Attempt the update operation
        table.update_item(
            Key=key,
            UpdateExpression=update_expression,
            ExpressionAttributeNames=expression_attribute_names,
            ExpressionAttributeValues=expression_attribute_values,
        )
        return True, None
    except ClientError as e:
        error_message = e.response["Error"]["Message"]

        if "expression contains too many operators" in error_message.lower():
            return False, error_message
        else:
            # Other error occurred
            raise
```

Example usage of expression operator counting with AWS SDK for Python (Boto3).

```
def example_usage():
    """Example of how to use the expression operator counting functions."""

    print("Example 1: Creating a complex filter expression with multiple
 conditions")
    attribute_name = "Status"
```

```
    values = ["Active", "Pending", "Processing", "Shipped", "Delivered"]

    filter_expr, expr_attr_vals, expr_attr_names, op_count =
create_complex_filter_expression(
        attribute_name=attribute_name, values=values, use_or=True
    )

    print(f"Filter Expression: {filter_expr}")
    print(f"Expression Attribute Values: {expr_attr_vals}")
    print(f"Expression Attribute Names: {expr_attr_names}")
    print(f"Operator Count: {op_count}")

    print("\nExample 2: Creating a nested filter expression")
    nested_expr, nested_vals, nested_names, nested_count =
create_nested_filter_expression(
        depth=3, conditions_per_level=2
    )

    print(f"Nested Filter Expression: {nested_expr}")
    print(f"Operator Count: {nested_count}")

    print("\nExample 3: Counting operators in an update expression")
    update_expression = "SET #name = :name, #age = :age + :increment,
#address.#city = :city, #status = if_not_exists(#status, :default_status) REMOVE
#old_field ADD #counter :value DELETE #set_attr :set_val"
    update_op_count = count_operators_in_update_expression(update_expression)

    print(f"Update Expression: {update_expression}")
    print(f"Operator Count: {update_op_count}")

    print("\nExample 4: Counting operators in a condition expression")
    condition_expression = "(#status = :active OR #status = :pending) AND #price
BETWEEN :min_price AND :max_price AND attribute_exists(#category) AND NOT
(#stock <= :min_stock)"
    condition_op_count =
count_operators_in_condition_expression(condition_expression)

    print(f"Condition Expression: {condition_expression}")
    print(f"Operator Count: {condition_op_count}")

    print("\nExample 5: Testing the 300 operator limit")

    # This is just for demonstration - in a real application, you would use your
actual table
```

```
    # Note: This function is renamed to _test_expression_limit to avoid pytest
trying to run it
    print("In a real application, you would test with _test_expression_limit
function")
    print("Expression with 250 operators would be under the limit")
    print("Expression with 350 operators would exceed the 300 operator limit")

    print("\nOperator Counting Rules in DynamoDB:")
    print("1. Comparison Operators (=, <>, <, <=, >, >=): 1 operator each")
    print("2. Logical Operators (AND, OR, NOT): 1 operator each")
    print("3. BETWEEN: 2 operators (BETWEEN + AND)")
    print("4. IN: 1 operator (regardless of number of values)")
    print("5. Functions (attribute_exists, begins_with, etc.): 1 operator each")
    print("6. Arithmetic Operators (+, -): 1 operator each")
    print("7. SET assignments (=): 1 operator each")
    print("8. ADD and DELETE operations: 1 operator each")

    print("\nStrategies for Working Within the 300 Operator Limit:")
    print("1. Break operations into multiple requests")
    print("2. Use DynamoDB Transactions for complex operations")
    print("3. Optimize data model to reduce query complexity")
    print("4. Use application-side filtering for less critical filters")
    print("5. Consider using IN operator instead of multiple OR conditions")
```

- For API details, see UpdateItem in *AWS SDK for Python (Boto3) API Reference*.

For a complete list of AWS SDK developer guides and code examples, see Using DynamoDB with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Create an API Gateway REST API to track COVID-19 data

The following code example shows how to create a REST API that simulates a system to track daily cases of COVID-19 in the United States, using fictional data.

Python

SDK for Python (Boto3)

Shows how to use AWS Chalice with the AWS SDK for Python (Boto3) to create a serverless REST API that uses Amazon API Gateway, AWS Lambda, and Amazon DynamoDB. The REST API simulates a system that tracks daily cases of COVID-19 in the United States, using fictional data. Learn how to:

- Use AWS Chalice to define routes in Lambda functions that are called to handle REST requests that come through API Gateway.
- Use Lambda functions to retrieve and store data in a DynamoDB table to serve REST requests.
- Define table structure and security role resources in an AWS CloudFormation template.
- Use AWS Chalice and CloudFormation to package and deploy all necessary resources.
- Use CloudFormation to clean up all created resources.

For complete source code and instructions on how to set up and run, see the full example on GitHub.

Services used in this example

- API Gateway
- CloudFormation
- DynamoDB
- Lambda

For a complete list of AWS SDK developer guides and code examples, see Using DynamoDB with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Create a messenger application with Step Functions

The following code example shows how to create an AWS Step Functions messenger application that retrieves message records from a database table.

Python

SDK for Python (Boto3)

Shows how to use the AWS SDK for Python (Boto3) with AWS Step Functions to create a messenger application that retrieves message records from an Amazon DynamoDB table and sends them with Amazon Simple Queue Service (Amazon SQS). The state machine integrates with an AWS Lambda function to scan the database for unsent messages.

- Create a state machine that retrieves and updates message records from an Amazon DynamoDB table.
- Update the state machine definition to also send messages to Amazon Simple Queue Service (Amazon SQS).
- Start and stop state machine runs.
- Connect to Lambda, DynamoDB, and Amazon SQS from a state machine by using service integrations.

For complete source code and instructions on how to set up and run, see the full example on GitHub.

Services used in this example

- DynamoDB
- Lambda
- Amazon SQS
- Step Functions

For a complete list of AWS SDK developer guides and code examples, see Using DynamoDB with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Create a photo asset management application that lets users manage photos using labels

The following code examples show how to create a serverless application that lets users manage photos using labels.

.NET

SDK for .NET

Shows how to develop a photo asset management application that detects labels in images using Amazon Rekognition and stores them for later retrieval.

For complete source code and instructions on how to set up and run, see the full example on GitHub.

For a deep dive into the origin of this example see the post on AWS Community.

Services used in this example

- API Gateway
- DynamoDB
- Lambda
- Amazon Rekognition
- Amazon S3
- Amazon SNS

C++

SDK for C++

Shows how to develop a photo asset management application that detects labels in images using Amazon Rekognition and stores them for later retrieval.

For complete source code and instructions on how to set up and run, see the full example on GitHub.

For a deep dive into the origin of this example see the post on AWS Community.

Services used in this example

- API Gateway
- DynamoDB
- Lambda
- Amazon Rekognition

- Amazon S3
- Amazon SNS

Java

SDK for Java 2.x

Shows how to develop a photo asset management application that detects labels in images using Amazon Rekognition and stores them for later retrieval.

For complete source code and instructions on how to set up and run, see the full example on GitHub.

For a deep dive into the origin of this example see the post on AWS Community.

Services used in this example

- API Gateway
- DynamoDB
- Lambda
- Amazon Rekognition
- Amazon S3
- Amazon SNS

JavaScript

SDK for JavaScript (v3)

Shows how to develop a photo asset management application that detects labels in images using Amazon Rekognition and stores them for later retrieval.

For complete source code and instructions on how to set up and run, see the full example on GitHub.

For a deep dive into the origin of this example see the post on AWS Community.

Services used in this example

- API Gateway

- DynamoDB
- Lambda
- Amazon Rekognition
- Amazon S3
- Amazon SNS

Kotlin

SDK for Kotlin

Shows how to develop a photo asset management application that detects labels in images using Amazon Rekognition and stores them for later retrieval.

For complete source code and instructions on how to set up and run, see the full example on GitHub.

For a deep dive into the origin of this example see the post on AWS Community.

Services used in this example

- API Gateway
- DynamoDB
- Lambda
- Amazon Rekognition
- Amazon S3
- Amazon SNS

PHP

SDK for PHP

Shows how to develop a photo asset management application that detects labels in images using Amazon Rekognition and stores them for later retrieval.

For complete source code and instructions on how to set up and run, see the full example on GitHub.

For a deep dive into the origin of this example see the post on AWS Community.

Services used in this example

- API Gateway
- DynamoDB
- Lambda
- Amazon Rekognition
- Amazon S3
- Amazon SNS

Rust

SDK for Rust

Shows how to develop a photo asset management application that detects labels in images using Amazon Rekognition and stores them for later retrieval.

For complete source code and instructions on how to set up and run, see the full example on GitHub.

For a deep dive into the origin of this example see the post on AWS Community.

Services used in this example

- API Gateway
- DynamoDB
- Lambda
- Amazon Rekognition
- Amazon S3
- Amazon SNS

For a complete list of AWS SDK developer guides and code examples, see Using DynamoDB with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Create a DynamoDB table with a Global Secondary Index using the AWS SDK

The following code example shows how to create a table with global secondary index.

Java

SDK for Java 2.x

Create DynamoDB table with Global Secondary Index using AWS SDK for Java 2.x.

```
import software.amazon.awssdk.auth.credentials.DefaultCredentialsProvider;
import software.amazon.awssdk.core.waiters.WaiterResponse;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeDefinition;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.CreateTableRequest;
import software.amazon.awssdk.services.dynamodb.model.DeleteTableRequest;
import software.amazon.awssdk.services.dynamodb.model.DescribeTableRequest;
import software.amazon.awssdk.services.dynamodb.model.DescribeTableResponse;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.GlobalSecondaryIndex;
import software.amazon.awssdk.services.dynamodb.model.KeySchemaElement;
import software.amazon.awssdk.services.dynamodb.model.KeyType;
import software.amazon.awssdk.services.dynamodb.model.Projection;
import software.amazon.awssdk.services.dynamodb.model.ProjectionType;
import software.amazon.awssdk.services.dynamodb.model.ProvisionedThroughput;
import software.amazon.awssdk.services.dynamodb.model.PutItemRequest;
import software.amazon.awssdk.services.dynamodb.model.QueryRequest;
import software.amazon.awssdk.services.dynamodb.model.QueryResponse;
import software.amazon.awssdk.services.dynamodb.model.ScalarAttributeType;
import software.amazon.awssdk.services.dynamodb.waiters.DynamoDbWaiter;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

    public void createTable() {
        try {
            // Attribute definitions
            final List<AttributeDefinition> attributeDefinitions = new
 ArrayList<>();
            attributeDefinitions.add(AttributeDefinition.builder()
                .attributeName(ISSUE_ID_ATTR)
                .attributeType(ScalarAttributeType.S)
                .build());
            attributeDefinitions.add(AttributeDefinition.builder()
```

```
                .attributeName(TITLE_ATTR)
                .attributeType(ScalarAttributeType.S)
                .build());
        attributeDefinitions.add(AttributeDefinition.builder()
                .attributeName(CREATE_DATE_ATTR)
                .attributeType(ScalarAttributeType.S)
                .build());
        attributeDefinitions.add(AttributeDefinition.builder()
                .attributeName(DUE_DATE_ATTR)
                .attributeType(ScalarAttributeType.S)
                .build());

        // Key schema for table
        final List<KeySchemaElement> tableKeySchema = new ArrayList<>();
        tableKeySchema.add(KeySchemaElement.builder()
                .attributeName(ISSUE_ID_ATTR)
                .keyType(KeyType.HASH)
                .build()); // Partition key
        tableKeySchema.add(KeySchemaElement.builder()
                .attributeName(TITLE_ATTR)
                .keyType(KeyType.RANGE)
                .build()); // Sort key

        // Initial provisioned throughput settings for the indexes
        final ProvisionedThroughput ptIndex = ProvisionedThroughput.builder()
                .readCapacityUnits(1L)
                .writeCapacityUnits(1L)
                .build();

        // CreateDateIndex
        final List<KeySchemaElement> createDateKeySchema = new ArrayList<>();
        createDateKeySchema.add(KeySchemaElement.builder()
                .attributeName(CREATE_DATE_ATTR)
                .keyType(KeyType.HASH)
                .build());
        createDateKeySchema.add(KeySchemaElement.builder()
                .attributeName(ISSUE_ID_ATTR)
                .keyType(KeyType.RANGE)
                .build());

        final Projection createDateProjection = Projection.builder()
                .projectionType(ProjectionType.INCLUDE)
                .nonKeyAttributes(DESCRIPTION_ATTR, STATUS_ATTR)
                .build();
```

```
            final GlobalSecondaryIndex createDateIndex =
GlobalSecondaryIndex.builder()
                .indexName(CREATE_DATE_INDEX)
                .keySchema(createDateKeySchema)
                .projection(createDateProjection)
                .provisionedThroughput(ptIndex)
                .build();

            // TitleIndex
            final List<KeySchemaElement> titleKeySchema = new ArrayList<>();
            titleKeySchema.add(KeySchemaElement.builder()
                .attributeName(TITLE_ATTR)
                .keyType(KeyType.HASH)
                .build());
            titleKeySchema.add(KeySchemaElement.builder()
                .attributeName(ISSUE_ID_ATTR)
                .keyType(KeyType.RANGE)
                .build());

            final Projection titleProjection =

Projection.builder().projectionType(ProjectionType.KEYS_ONLY).build();

            final GlobalSecondaryIndex titleIndex =
GlobalSecondaryIndex.builder()
                .indexName(TITLE_INDEX)
                .keySchema(titleKeySchema)
                .projection(titleProjection)
                .provisionedThroughput(ptIndex)
                .build();

            // DueDateIndex
            final List<KeySchemaElement> dueDateKeySchema = new ArrayList<>();
            dueDateKeySchema.add(KeySchemaElement.builder()
                .attributeName(DUE_DATE_ATTR)
                .keyType(KeyType.HASH)
                .build());

            final Projection dueDateProjection =
                Projection.builder().projectionType(ProjectionType.ALL).build();

            final GlobalSecondaryIndex dueDateIndex =
GlobalSecondaryIndex.builder()
```

```
                .indexName(DUE_DATE_INDEX)
                .keySchema(dueDateKeySchema)
                .projection(dueDateProjection)
                .provisionedThroughput(ptIndex)
                .build();

            final CreateTableRequest createTableRequest =
CreateTableRequest.builder()
                .tableName(TABLE_NAME)
                .keySchema(tableKeySchema)
                .attributeDefinitions(attributeDefinitions)
                .globalSecondaryIndexes(createDateIndex, titleIndex,
dueDateIndex)
                .provisionedThroughput(ProvisionedThroughput.builder()
                    .readCapacityUnits(1L)
                    .writeCapacityUnits(1L)
                    .build())
                .build();

            System.out.println("Creating table " + TABLE_NAME + "...");
            dynamoDbClient.createTable(createTableRequest);

            // Wait for table to become active
            System.out.println("Waiting for " + TABLE_NAME + " to become
ACTIVE...");
            final DynamoDbWaiter waiter = dynamoDbClient.waiter();
            final DescribeTableRequest describeTableRequest =
                DescribeTableRequest.builder().tableName(TABLE_NAME).build();

            final WaiterResponse<DescribeTableResponse> waiterResponse =
                waiter.waitUntilTableExists(describeTableRequest);
            waiterResponse.matched().response().ifPresent(response ->
System.out.println("Table is now ready for use"));

        } catch (DynamoDbException e) {
            System.err.println("Error creating table: " + e.getMessage());
            e.printStackTrace();
        }
    }
```

- For API details, see CreateTable in *AWS SDK for Java 2.x API Reference*.

For a complete list of AWS SDK developer guides and code examples, see Using DynamoDB with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Create a DynamoDB table with warm throughput setting using an AWS SDK

The following code examples show how to create a table with warm throughput enabled.

Java

SDK for Java 2.x

Create DynamoDB table with warm throughput setting using AWS SDK for Java 2.x.

```
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeDefinition;
import software.amazon.awssdk.services.dynamodb.model.CreateTableRequest;
import software.amazon.awssdk.services.dynamodb.model.CreateTableResponse;
import software.amazon.awssdk.services.dynamodb.model.GlobalSecondaryIndex;
import software.amazon.awssdk.services.dynamodb.model.KeySchemaElement;
import software.amazon.awssdk.services.dynamodb.model.KeyType;
import software.amazon.awssdk.services.dynamodb.model.Projection;
import software.amazon.awssdk.services.dynamodb.model.ProvisionedThroughput;
import software.amazon.awssdk.services.dynamodb.model.ScalarAttributeType;
import software.amazon.awssdk.services.dynamodb.model.WarmThroughput;

    public static WarmThroughput buildWarmThroughput(final Long
 readUnitsPerSecond, final Long writeUnitsPerSecond) {
        return WarmThroughput.builder()
            .readUnitsPerSecond(readUnitsPerSecond)
            .writeUnitsPerSecond(writeUnitsPerSecond)
            .build();
    }

    /**
     * Builds a ProvisionedThroughput object with the specified read and write
 capacity units.
     *
     * @param readCapacityUnits The read capacity units
     * @param writeCapacityUnits The write capacity units
     * @return A configured ProvisionedThroughput object
     */
```

```
    public static ProvisionedThroughput buildProvisionedThroughput(
        final Long readCapacityUnits, final Long writeCapacityUnits) {
        return ProvisionedThroughput.builder()
            .readCapacityUnits(readCapacityUnits)
            .writeCapacityUnits(writeCapacityUnits)
            .build();
    }

    /**
     * Builds an AttributeDefinition with the specified name and type.
     *
     * @param attributeName The attribute name
     * @param scalarAttributeType The attribute type
     * @return A configured AttributeDefinition
     */
    private static AttributeDefinition buildAttributeDefinition(
        final String attributeName, final ScalarAttributeType
scalarAttributeType) {
        return AttributeDefinition.builder()
            .attributeName(attributeName)
            .attributeType(scalarAttributeType)
            .build();
    }

    /**
     * Builds a KeySchemaElement with the specified name and key type.
     *
     * @param attributeName The attribute name
     * @param keyType The key type (HASH or RANGE)
     * @return A configured KeySchemaElement
     */
    private static KeySchemaElement buildKeySchemaElement(final String
attributeName, final KeyType keyType) {
        return KeySchemaElement.builder()
            .attributeName(attributeName)
            .keyType(keyType)
            .build();
    }

    /**
     * Creates a DynamoDB table with the specified configuration including warm
throughput settings.
     *
     * @param ddb The DynamoDB client
```

```
    * @param tableName The name of the table to create
    * @param partitionKey The partition key attribute name
    * @param sortKey The sort key attribute name
    * @param miscellaneousKeyAttribute Additional key attribute name for GSI
    * @param nonKeyAttribute Non-key attribute to include in GSI projection
    * @param tableReadCapacityUnits Read capacity units for the table
    * @param tableWriteCapacityUnits Write capacity units for the table
    * @param tableWarmReadUnitsPerSecond Warm read units per second for the
table
    * @param tableWarmWriteUnitsPerSecond Warm write units per second for the
table
    * @param globalSecondaryIndexName The name of the GSI to create
    * @param globalSecondaryIndexReadCapacityUnits Read capacity units for the
GSI
    * @param globalSecondaryIndexWriteCapacityUnits Write capacity units for the
GSI
    * @param globalSecondaryIndexWarmReadUnitsPerSecond Warm read units per
second for the GSI
    * @param globalSecondaryIndexWarmWriteUnitsPerSecond Warm write units per
second for the GSI
    */
   public static void createDynamoDBTable(
       final DynamoDbClient ddb,
       final String tableName,
       final String partitionKey,
       final String sortKey,
       final String miscellaneousKeyAttribute,
       final String nonKeyAttribute,
       final Long tableReadCapacityUnits,
       final Long tableWriteCapacityUnits,
       final Long tableWarmReadUnitsPerSecond,
       final Long tableWarmWriteUnitsPerSecond,
       final String globalSecondaryIndexName,
       final Long globalSecondaryIndexReadCapacityUnits,
       final Long globalSecondaryIndexWriteCapacityUnits,
       final Long globalSecondaryIndexWarmReadUnitsPerSecond,
       final Long globalSecondaryIndexWarmWriteUnitsPerSecond) {

       // Define the table attributes
       final AttributeDefinition partitionKeyAttribute =
buildAttributeDefinition(partitionKey, ScalarAttributeType.S);
       final AttributeDefinition sortKeyAttribute =
buildAttributeDefinition(sortKey, ScalarAttributeType.S);
       final AttributeDefinition miscellaneousKeyAttributeDefinition =
```

```
            buildAttributeDefinition(miscellaneousKeyAttribute,
ScalarAttributeType.N);
        final AttributeDefinition[] attributeDefinitions = {
            partitionKeyAttribute, sortKeyAttribute,
miscellaneousKeyAttributeDefinition
        };

        // Define the table key schema
        final KeySchemaElement partitionKeyElement =
buildKeySchemaElement(partitionKey, KeyType.HASH);
        final KeySchemaElement sortKeyElement = buildKeySchemaElement(sortKey,
KeyType.RANGE);
        final KeySchemaElement[] keySchema = {partitionKeyElement,
sortKeyElement};

        // Define the provisioned throughput for the table
        final ProvisionedThroughput provisionedThroughput =
            buildProvisionedThroughput(tableReadCapacityUnits,
tableWriteCapacityUnits);

        // Define the Global Secondary Index (GSI)
        final KeySchemaElement globalSecondaryIndexPartitionKeyElement =
buildKeySchemaElement(sortKey, KeyType.HASH);
        final KeySchemaElement globalSecondaryIndexSortKeyElement =
            buildKeySchemaElement(miscellaneousKeyAttribute, KeyType.RANGE);
        final KeySchemaElement[] gsiKeySchema = {
            globalSecondaryIndexPartitionKeyElement,
globalSecondaryIndexSortKeyElement
        };

        final Projection gsiProjection = Projection.builder()
            .projectionType(PROJECTION_TYPE_INCLUDE)
            .nonKeyAttributes(nonKeyAttribute)
            .build();

        final ProvisionedThroughput gsiProvisionedThroughput =
            buildProvisionedThroughput(globalSecondaryIndexReadCapacityUnits,
globalSecondaryIndexWriteCapacityUnits);

        // Define the warm throughput for the Global Secondary Index (GSI)
        final WarmThroughput gsiWarmThroughput = buildWarmThroughput(
            globalSecondaryIndexWarmReadUnitsPerSecond,
globalSecondaryIndexWarmWriteUnitsPerSecond);
```

```
        final GlobalSecondaryIndex globalSecondaryIndex =
 GlobalSecondaryIndex.builder()
            .indexName(globalSecondaryIndexName)
            .keySchema(gsiKeySchema)
            .projection(gsiProjection)
            .provisionedThroughput(gsiProvisionedThroughput)
            .warmThroughput(gsiWarmThroughput)
            .build();

        // Define the warm throughput for the table
        final WarmThroughput tableWarmThroughput =
            buildWarmThroughput(tableWarmReadUnitsPerSecond,
 tableWarmWriteUnitsPerSecond);

        final CreateTableRequest request = CreateTableRequest.builder()
            .tableName(tableName)
            .attributeDefinitions(attributeDefinitions)
            .keySchema(keySchema)
            .provisionedThroughput(provisionedThroughput)
            .globalSecondaryIndexes(globalSecondaryIndex)
            .warmThroughput(tableWarmThroughput)
            .build();

        final CreateTableResponse response = ddb.createTable(request);
        System.out.println(response);
    }
```

- For API details, see [CreateTable](#) in *AWS SDK for Java 2.x API Reference*.

JavaScript

SDK for JavaScript (v3)

Create DynamoDB table with warm throughput setting using AWS SDK for JavaScript.

```
import { DynamoDBClient, CreateTableCommand } from "@aws-sdk/client-dynamodb";

export async function createDynamoDBTableWithWarmThroughput(
  tableName,
  partitionKey,
  sortKey,
  miscKeyAttr,
```

```
  nonKeyAttr,
  tableProvisionedReadUnits,
  tableProvisionedWriteUnits,
  tableWarmReads,
  tableWarmWrites,
  indexName,
  indexProvisionedReadUnits,
  indexProvisionedWriteUnits,
  indexWarmReads,
  indexWarmWrites,
  region = "us-east-1"
) {
  try {
    const ddbClient = new DynamoDBClient({ region: region });
    const command = new CreateTableCommand({
      TableName: tableName,
      AttributeDefinitions: [
          { AttributeName: partitionKey, AttributeType: "S" },
          { AttributeName: sortKey, AttributeType: "S" },
          { AttributeName: miscKeyAttr, AttributeType: "N" },
      ],
      KeySchema: [
          { AttributeName: partitionKey, KeyType: "HASH" },
          { AttributeName: sortKey, KeyType: "RANGE" },
      ],
      ProvisionedThroughput: {
          ReadCapacityUnits: tableProvisionedReadUnits,
          WriteCapacityUnits: tableProvisionedWriteUnits,
      },
      WarmThroughput: {
          ReadUnitsPerSecond: tableWarmReads,
          WriteUnitsPerSecond: tableWarmWrites,
      },
      GlobalSecondaryIndexes: [
          {
            IndexName: indexName,
            KeySchema: [
                { AttributeName: sortKey, KeyType: "HASH" },
                { AttributeName: miscKeyAttr, KeyType: "RANGE" },
            ],
            Projection: {
                ProjectionType: "INCLUDE",
                NonKeyAttributes: [nonKeyAttr],
            },
```

```
            ProvisionedThroughput: {
                ReadCapacityUnits: indexProvisionedReadUnits,
                WriteCapacityUnits: indexProvisionedWriteUnits,
            },
            WarmThroughput: {
                ReadUnitsPerSecond: indexWarmReads,
                WriteUnitsPerSecond: indexWarmWrites,
            },
          },
      ],
    });
    const response = await ddbClient.send(command);
    console.log(response);
    return response;
  } catch (error) {
    console.error(`Error creating table: ${error}`);
    throw error;
  }
}

// Example usage (commented out for testing)
/*
createDynamoDBTableWithWarmThroughput(
  'example-table',
  'pk',
  'sk',
  'gsiKey',
  'data',
  10, 10, 5, 5,
  'example-index',
  5, 5, 2, 2
);
*/
```

- For API details, see CreateTable in *AWS SDK for JavaScript API Reference*.

Python

SDK for Python (Boto3)

Create DynamoDB table with warm throughput setting using AWS SDK for Python (Boto3).

```
from boto3 import client
from botocore.exceptions import ClientError


def create_dynamodb_table_warm_throughput(
    table_name,
    partition_key,
    sort_key,
    misc_key_attr,
    non_key_attr,
    table_provisioned_read_units,
    table_provisioned_write_units,
    table_warm_reads,
    table_warm_writes,
    gsi_name,
    gsi_provisioned_read_units,
    gsi_provisioned_write_units,
    gsi_warm_reads,
    gsi_warm_writes,
    region_name="us-east-1",
):
    """
    Creates a DynamoDB table with a warm throughput setting configured.

    :param table_name: The name of the table to be created.
    :param partition_key: The partition key for the table being created.
    :param sort_key: The sort key for the table being created.
    :param misc_key_attr: A miscellaneous key attribute for the table being
 created.
    :param non_key_attr: A non-key attribute for the table being created.
    :param table_provisioned_read_units: The newly created table's provisioned
 read capacity units.
    :param table_provisioned_write_units: The newly created table's provisioned
 write capacity units.
    :param table_warm_reads: The read units per second setting for the table's
 warm throughput.
    :param table_warm_writes: The write units per second setting for the table's
 warm throughput.
    :param gsi_name: The name of the Global Secondary Index (GSI) to be created
 on the table.
    :param gsi_provisioned_read_units: The configured Global Secondary Index
 (GSI) provisioned read capacity units.
```

```
   :param gsi_provisioned_write_units: The configured Global Secondary Index
(GSI) provisioned write capacity units.
   :param gsi_warm_reads: The read units per second setting for the Global
Secondary Index (GSI)'s warm throughput.
   :param gsi_warm_writes: The write units per second setting for the Global
Secondary Index (GSI)'s warm throughput.
   :param region_name: The AWS Region name to target. defaults to us-east-1
   """
   try:
       ddb = client("dynamodb", region_name=region_name)

       # Define the table attributes
       attribute_definitions = [
           {"AttributeName": partition_key, "AttributeType": "S"},
           {"AttributeName": sort_key, "AttributeType": "S"},
           {"AttributeName": misc_key_attr, "AttributeType": "N"},
       ]

       # Define the table key schema
       key_schema = [
           {"AttributeName": partition_key, "KeyType": "HASH"},
           {"AttributeName": sort_key, "KeyType": "RANGE"},
       ]

       # Define the provisioned throughput for the table
       provisioned_throughput = {
           "ReadCapacityUnits": table_provisioned_read_units,
           "WriteCapacityUnits": table_provisioned_write_units,
       }

       # Define the global secondary index
       gsi_key_schema = [
           {"AttributeName": sort_key, "KeyType": "HASH"},
           {"AttributeName": misc_key_attr, "KeyType": "RANGE"},
       ]
       gsi_projection = {"ProjectionType": "INCLUDE", "NonKeyAttributes":
[non_key_attr]}
       gsi_provisioned_throughput = {
           "ReadCapacityUnits": gsi_provisioned_read_units,
           "WriteCapacityUnits": gsi_provisioned_write_units,
       }
       gsi_warm_throughput = {
           "ReadUnitsPerSecond": gsi_warm_reads,
           "WriteUnitsPerSecond": gsi_warm_writes,
```

```
        }
        global_secondary_indexes = [
            {
                "IndexName": gsi_name,
                "KeySchema": gsi_key_schema,
                "Projection": gsi_projection,
                "ProvisionedThroughput": gsi_provisioned_throughput,
                "WarmThroughput": gsi_warm_throughput,
            }
        ]

        # Define the warm throughput for the table
        warm_throughput = {
            "ReadUnitsPerSecond": table_warm_reads,
            "WriteUnitsPerSecond": table_warm_writes,
        }

        # Create the DynamoDB client and create the table
        response = ddb.create_table(
            TableName=table_name,
            AttributeDefinitions=attribute_definitions,
            KeySchema=key_schema,
            ProvisionedThroughput=provisioned_throughput,
            GlobalSecondaryIndexes=global_secondary_indexes,
            WarmThroughput=warm_throughput,
        )

        print(response)
        return response
    except ClientError as e:
        print(f"Error creating table: {e}")
        raise e
```

- For API details, see CreateTable in *AWS SDK for Python (Boto3) API Reference*.

For a complete list of AWS SDK developer guides and code examples, see Using DynamoDB with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Create a web application to track DynamoDB data

The following code examples show how to create a web application that tracks work items in an Amazon DynamoDB table and uses Amazon Simple Email Service (Amazon SES) to send reports.

.NET

SDK for .NET

Shows how to use the Amazon DynamoDB .NET API to create a dynamic web application that tracks DynamoDB work data.

For complete source code and instructions on how to set up and run, see the full example on GitHub.

Services used in this example

- DynamoDB
- Amazon SES

Java

SDK for Java 2.x

Shows how to use the Amazon DynamoDB API to create a dynamic web application that tracks DynamoDB work data.

For complete source code and instructions on how to set up and run, see the full example on GitHub.

Services used in this example

- DynamoDB
- Amazon SES

Kotlin

SDK for Kotlin

Shows how to use the Amazon DynamoDB API to create a dynamic web application that tracks DynamoDB work data.

For complete source code and instructions on how to set up and run, see the full example on GitHub.

Services used in this example

- DynamoDB
- Amazon SES

Python

SDK for Python (Boto3)

Shows how to use the AWS SDK for Python (Boto3) to create a REST service that tracks work items in Amazon DynamoDB and emails reports by using Amazon Simple Email Service (Amazon SES). This example uses the Flask web framework to handle HTTP routing and integrates with a React webpage to present a fully functional web application.

- Build a Flask REST service that integrates with AWS services.
- Read, write, and update work items that are stored in a DynamoDB table.
- Use Amazon SES to send email reports of work items.

For complete source code and instructions on how to set up and run, see the full example in the AWS Code Examples Repository on GitHub.

Services used in this example

- DynamoDB
- Amazon SES

For a complete list of AWS SDK developer guides and code examples, see Using DynamoDB with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Create a websocket chat application with API Gateway

The following code example shows how to create a chat application that is served by a websocket API built on Amazon API Gateway.

Python

SDK for Python (Boto3)

Shows how to use the AWS SDK for Python (Boto3) with Amazon API Gateway V2 to create a websocket API that integrates with AWS Lambda and Amazon DynamoDB.

- Create a websocket API served by API Gateway.
- Define a Lambda handler that stores connections in DynamoDB and posts messages to other chat participants.
- Connect to the websocket chat application and send messages with the Websockets package.

For complete source code and instructions on how to set up and run, see the full example on GitHub.

Services used in this example

- API Gateway
- DynamoDB
- Lambda

For a complete list of AWS SDK developer guides and code examples, see Using DynamoDB with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Create a DynamoDB item with a TTL using an AWS SDK

The following code examples show how to create an item with TTL.

Java

SDK for Java 2.x

```
package com.amazon.samplelib.ttl;

import com.amazon.samplelib.CodeSampleUtils;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
```

```
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.PutItemRequest;
import software.amazon.awssdk.services.dynamodb.model.PutItemResponse;
import software.amazon.awssdk.services.dynamodb.model.ResourceNotFoundException;

import java.util.HashMap;
import java.util.Map;
import java.util.Optional;

/**
 * Creates an item in a DynamoDB table with TTL attributes.
 * This class demonstrates how to add TTL expiration timestamps to DynamoDB
 items.
 */
public class CreateTTL {

    private static final String USAGE =
        """
            Usage:
                <tableName> <primaryKey> <sortKey> <region>
            Where:
                tableName - The Amazon DynamoDB table being queried.
                primaryKey - The name of the primary key. Also known as the hash
 or partition key.
                sortKey - The name of the sort key. Also known as the range
 attribute.
                region (optional) - The AWS region that the Amazon DynamoDB table
 is located in. (Default: us-east-1)
            """;
    private static final int DAYS_TO_EXPIRE = 90;
    private static final int SECONDS_PER_DAY = 24 * 60 * 60;
    private static final String PRIMARY_KEY_ATTR = "primaryKey";
    private static final String SORT_KEY_ATTR = "sortKey";
    private static final String CREATION_DATE_ATTR = "creationDate";
    private static final String EXPIRE_AT_ATTR = "expireAt";
    private static final String SUCCESS_MESSAGE = "%s PutItem operation with TTL
 successful.";
    private static final String TABLE_NOT_FOUND_ERROR = "Error: The Amazon
 DynamoDB table \"%s\" can't be found.";

    private final DynamoDbClient dynamoDbClient;

    /**
     * Constructs a CreateTTL instance with the specified DynamoDB client.
```

```
     *
     * @param dynamoDbClient The DynamoDB client to use
     */
    public CreateTTL(final DynamoDbClient dynamoDbClient) {
        this.dynamoDbClient = dynamoDbClient;
    }

    /**
     * Constructs a CreateTTL with a default DynamoDB client.
     */
    public CreateTTL() {
        this.dynamoDbClient = null;
    }

    /**
     * Main method to demonstrate creating an item with TTL.
     *
     * @param args Command line arguments
     */
    public static void main(final String[] args) {
        try {
            int result = new CreateTTL().processArgs(args);
            System.exit(result);
        } catch (Exception e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }

    /**
     * Process command line arguments and create an item with TTL.
     *
     * @param args Command line arguments
     * @return 0 if successful, non-zero otherwise
     * @throws ResourceNotFoundException If the table doesn't exist
     * @throws DynamoDbException If an error occurs during the operation
     * @throws IllegalArgumentException If arguments are invalid
     */
    public int processArgs(final String[] args) {
        // Argument validation (remove or replace this line when reusing this
code)
        CodeSampleUtils.validateArgs(args, new int[] {3, 4}, USAGE);

        final String tableName = args[0];
```

```
        final String primaryKey = args[1];
        final String sortKey = args[2];
        final Region region = Optional.ofNullable(args.length > 3 ? args[3] :
null)
            .map(Region::of)
            .orElse(Region.US_EAST_1);

        try (DynamoDbClient ddb = dynamoDbClient != null
            ? dynamoDbClient
            : DynamoDbClient.builder().region(region).build()) {
            final CreateTTL createTTL = new CreateTTL(ddb);
            createTTL.createItemWithTTL(tableName, primaryKey, sortKey);
            return 0;
        } catch (Exception e) {
            throw e;
        }
    }

    /**
     * Creates an item in the specified table with TTL attributes.
     *
     * @param tableName The name of the table
     * @param primaryKeyValue The value for the primary key
     * @param sortKeyValue The value for the sort key
     * @return The response from the PutItem operation
     * @throws ResourceNotFoundException If the table doesn't exist
     * @throws DynamoDbException If an error occurs during the operation
     */
    public PutItemResponse createItemWithTTL(
        final String tableName, final String primaryKeyValue, final String
sortKeyValue) {
        // Get current time in epoch second format
        final long createDate = System.currentTimeMillis() / 1000;

        // Calculate expiration time 90 days from now in epoch second format
        final long expireDate = createDate + (DAYS_TO_EXPIRE * SECONDS_PER_DAY);

        final Map<String, AttributeValue> itemMap = new HashMap<>();
        itemMap.put(
            PRIMARY_KEY_ATTR,
AttributeValue.builder().s(primaryKeyValue).build());
        itemMap.put(SORT_KEY_ATTR,
AttributeValue.builder().s(sortKeyValue).build());
        itemMap.put(
```

```
            CREATION_DATE_ATTR,
            AttributeValue.builder().n(String.valueOf(createDate)).build());
        itemMap.put(
            EXPIRE_AT_ATTR,
            AttributeValue.builder().n(String.valueOf(expireDate)).build());

        final PutItemRequest request =
            PutItemRequest.builder().tableName(tableName).item(itemMap).build();

        try {
            final PutItemResponse response = dynamoDbClient.putItem(request);
            System.out.println(String.format(SUCCESS_MESSAGE, tableName));
            return response;
        } catch (ResourceNotFoundException e) {
            System.err.format(TABLE_NOT_FOUND_ERROR, tableName);
            throw e;
        } catch (DynamoDbException e) {
            System.err.println(e.getMessage());
            throw e;
        }
    }
}
```

- For API details, see PutItem in *AWS SDK for Java 2.x API Reference*.

JavaScript

SDK for JavaScript (v3)

```
import { DynamoDBClient, PutItemCommand } from "@aws-sdk/client-dynamodb";

export function createDynamoDBItem(table_name, region, partition_key, sort_key) {
    const client = new DynamoDBClient({
        region: region,
        endpoint: `https://dynamodb.${region}.amazonaws.com`
    });

    // Get the current time in epoch second format
    const current_time = Math.floor(new Date().getTime() / 1000);

    // Calculate the expireAt time (90 days from now) in epoch second format
```

```
    const expire_at = Math.floor((new Date().getTime() + 90 * 24 * 60 * 60 *
 1000) / 1000);

    // Create DynamoDB item
    const item = {
        'partitionKey': {'S': partition_key},
        'sortKey': {'S': sort_key},
        'createdAt': {'N': current_time.toString()},
        'expireAt': {'N': expire_at.toString()}
    };

    const putItemCommand = new PutItemCommand({
        TableName: table_name,
        Item: item,
        ProvisionedThroughput: {
            ReadCapacityUnits: 1,
            WriteCapacityUnits: 1,
        },
    });

    client.send(putItemCommand, function(err, data) {
        if (err) {
            console.log("Exception encountered when creating item %s, here's what
 happened: ", data, err);
            throw err;
        } else {
            console.log("Item created successfully: %s.", data);
            return data;
        }
    });
}

// Example usage (commented out for testing)
// createDynamoDBItem('your-table-name', 'us-east-1', 'your-partition-key-value',
 'your-sort-key-value');
```

- For API details, see PutItem in *AWS SDK for JavaScript API Reference*.

Python

SDK for Python (Boto3)

```
from datetime import datetime, timedelta
```

```
import boto3


def create_dynamodb_item(table_name, region, primary_key, sort_key):
    """
    Creates a DynamoDB item with an attached expiry attribute.

    :param table_name: Table name for the boto3 resource to target when creating
 an item
    :param region: string representing the AWS region. Example: `us-east-1`
    :param primary_key: one attribute known as the partition key.
    :param sort_key: Also known as a range attribute.
    :return: Void (nothing)
    """
    try:
        dynamodb = boto3.resource("dynamodb", region_name=region)
        table = dynamodb.Table(table_name)

        # Get the current time in epoch second format
        current_time = int(datetime.now().timestamp())

        # Calculate the expiration time (90 days from now) in epoch second format
        expiration_time = int((datetime.now() + timedelta(days=90)).timestamp())

        item = {
            "primaryKey": primary_key,
            "sortKey": sort_key,
            "creationDate": current_time,
            "expireAt": expiration_time,
        }
        response = table.put_item(Item=item)

        print("Item created successfully.")
        return response
    except Exception as e:
        print(f"Error creating item: {e}")
        raise e


# Use your own values
create_dynamodb_item(
    "your-table-name", "us-west-2", "your-partition-key-value", "your-sort-key-
value"
```

```
)
```

- For API details, see PutItem in *AWS SDK for Python (Boto3) API Reference*.

For a complete list of AWS SDK developer guides and code examples, see Using DynamoDB with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Create and manage DynamoDB global tables with Multi-Region Strong Consistency using an AWS SDK

The following code examples show how to create and manage DynamoDB global tables with Multi-Region Strong Consistency (MRSC).

- Create a table with Multi-Region Strong Consistency.
- Verify MRSC configuration and replica status.
- Test strong consistency across Regions with immediate reads.
- Perform conditional writes with MRSC guarantees.
- Clean up MRSC global table resources.

Bash

AWS CLI with Bash script

Create a table with Multi-Region Strong Consistency.

```
# Step 1: Create a new table in us-east-2 (primary region for MRSC)
# Note: Table must be empty when enabling MRSC
aws dynamodb create-table \
    --table-name MusicTable \
    --attribute-definitions \
        AttributeName=Artist,AttributeType=S \
        AttributeName=SongTitle,AttributeType=S \
    --key-schema \
        AttributeName=Artist,KeyType=HASH \
        AttributeName=SongTitle,KeyType=RANGE \
    --billing-mode PAY_PER_REQUEST \
```

```
    --region us-east-2

# Wait for table to become active
aws dynamodb wait table-exists --table-name MusicTable --region us-east-2

# Step 2: Add replica and witness with Multi-Region Strong Consistency
# MRSC requires exactly three replicas in supported regions
aws dynamodb update-table \
    --table-name MusicTable \
    --replica-updates '[{"Create": {"RegionName": "us-east-1"}}]' \
    --global-table-witness-updates '[{"Create": {"RegionName": "us-west-2"}}]' \
    --multi-region-consistency STRONG \
    --region us-east-2
```

Verify MRSC configuration and replica status.

```
# Verify the global table configuration and MRSC setting
aws dynamodb describe-table \
    --table-name MusicTable \
    --region us-east-2 \
    --query 'Table.
{TableName:TableName,TableStatus:TableStatus,MultiRegionConsistency:MultiRegionConsistenc
{Region:RegionName,Status:ReplicaStatus}}'
```

Test strong consistency with immediate reads across Regions.

```
# Write an item to the primary region
aws dynamodb put-item \
    --table-name MusicTable \
    --item '{"Artist": {"S":"The Beatles"},"SongTitle": {"S":"Hey Jude"},"Album":
 {"S":"The Beatles 1967-1970"},"Year": {"N":"1968"}}' \
    --region us-east-2

# Read the item from replica region to verify strong consistency (cannot read or
 write to witness)
# No wait time needed - MRSC provides immediate consistency
echo "Reading from us-east-1 (immediate consistency):"
aws dynamodb get-item \
    --table-name MusicTable \
    --key '{"Artist": {"S":"The Beatles"},"SongTitle": {"S":"Hey Jude"}}' \
    --consistent-read \
```

```
    --region us-east-1
```

Perform conditional writes with MRSC guarantees.

```
# Perform a conditional update from a different region
# This demonstrates that conditions work consistently across all regions
aws dynamodb update-item \
    --table-name MusicTable \
    --key '{"Artist": {"S":"The Beatles"},"SongTitle": {"S":"Hey Jude"}}' \
    --update-expression "SET #rating = :rating" \
    --condition-expression "attribute_exists(Artist)" \
    --expression-attribute-names '{"#rating": "Rating"}' \
    --expression-attribute-values '{":rating": {"N":"5"}}' \
    --region us-east-1
```

Clean up MRSC global table resources.

```
# Remove replica tables (must be done before deleting the primary table)
aws dynamodb update-table \
    --table-name MusicTable \
    --replica-updates '[{"Delete": {"RegionName": "us-east-1"}}]' \
    --global-table-witness-updates '[{"Delete": {"RegionName": "us-west-2"}}]' \
    --region us-east-2

# Wait for replicas to be deleted
echo "Waiting for replicas to be deleted..."
sleep 30

# Delete the primary table
aws dynamodb delete-table \
    --table-name MusicTable \
    --region us-east-2
```

- For API details, see the following topics in *AWS CLI Command Reference*.
  - CreateTable
  - DeleteTable
  - DescribeTable
  - GetItem

- PutItem
- UpdateItem
- UpdateTable

Java

SDK for Java 2.x

Create a regional table ready for MRSC conversion using AWS SDK for Java 2.x.

```
   public static CreateTableResponse createRegionalTable(final DynamoDbClient
dynamoDbClient, final String tableName) {

       if (dynamoDbClient == null) {
           throw new IllegalArgumentException("DynamoDB client cannot be null");
       }
       if (tableName == null || tableName.trim().isEmpty()) {
           throw new IllegalArgumentException("Table name cannot be null or
empty");
       }

       try {
           LOGGER.info("Creating regional table: " + tableName + " (must be
empty for MRSC)");

           CreateTableRequest createTableRequest = CreateTableRequest.builder()
               .tableName(tableName)
               .attributeDefinitions(
                   AttributeDefinition.builder()
                       .attributeName("Artist")
                       .attributeType(ScalarAttributeType.S)
                       .build(),
                   AttributeDefinition.builder()
                       .attributeName("SongTitle")
                       .attributeType(ScalarAttributeType.S)
                       .build())
               .keySchema(
                   KeySchemaElement.builder()
                       .attributeName("Artist")
                       .keyType(KeyType.HASH)
                       .build(),
                   KeySchemaElement.builder()
```

```
                        .attributeName("SongTitle")
                        .keyType(KeyType.RANGE)
                        .build())
                .billingMode(BillingMode.PAY_PER_REQUEST)
                .build();

            CreateTableResponse response =
dynamoDbClient.createTable(createTableRequest);
            LOGGER.info("Regional table creation initiated. Status: "
                + response.tableDescription().tableStatus());

            return response;

        } catch (DynamoDbException e) {
            LOGGER.severe("Failed to create regional table: " + tableName + " - "
+ e.getMessage());
            throw DynamoDbException.builder()
                .message("Failed to create regional table: " + tableName)
                .cause(e)
                .build();
        }
    }
```

Convert a regional table to MRSC with replicas and witness using AWS SDK for Java 2.x.

```
    public static UpdateTableResponse convertToMRSCWithWitness(
        final DynamoDbClient dynamoDbClient,
        final String tableName,
        final Region replicaRegion,
        final Region witnessRegion) {

        if (dynamoDbClient == null) {
            throw new IllegalArgumentException("DynamoDB client cannot be null");
        }
        if (tableName == null || tableName.trim().isEmpty()) {
            throw new IllegalArgumentException("Table name cannot be null or
empty");
        }
        if (replicaRegion == null) {
            throw new IllegalArgumentException("Replica region cannot be null");
        }
        if (witnessRegion == null) {
```

```
            throw new IllegalArgumentException("Witness region cannot be null");
        }

        try {
            LOGGER.info("Converting table to MRSC with replica in " +
replicaRegion.id() + " and witness in "
                + witnessRegion.id());

            // Create replica update using ReplicationGroupUpdate
            ReplicationGroupUpdate replicaUpdate =
ReplicationGroupUpdate.builder()
                .create(CreateReplicationGroupMemberAction.builder()
                    .regionName(replicaRegion.id())
                    .build())
                .build();

            // Create witness update
            GlobalTableWitnessGroupUpdate witnessUpdate =
GlobalTableWitnessGroupUpdate.builder()
                .create(CreateGlobalTableWitnessGroupMemberAction.builder()
                    .regionName(witnessRegion.id())
                    .build())
                .build();

            UpdateTableRequest updateTableRequest = UpdateTableRequest.builder()
                .tableName(tableName)
                .replicaUpdates(List.of(replicaUpdate))
                .globalTableWitnessUpdates(List.of(witnessUpdate))
                .multiRegionConsistency(MultiRegionConsistency.STRONG)
                .build();

            UpdateTableResponse response =
dynamoDbClient.updateTable(updateTableRequest);
            LOGGER.info("MRSC conversion initiated. Status: "
                + response.tableDescription().tableStatus());
            LOGGER.info("UpdateTableResponse full object: " + response);
            return response;

        } catch (DynamoDbException e) {
            LOGGER.severe("Failed to convert table to MRSC: " + tableName + " - "
+ e.getMessage());
            throw DynamoDbException.builder()
                .message("Failed to convert table to MRSC: " + tableName)
                .cause(e)
```

```
                .build();
        }
    }
```

Describe an MRSC global table configuration using AWS SDK for Java 2.x.

```
   public static DescribeTableResponse describeMRSCTable(final DynamoDbClient
dynamoDbClient, final String tableName) {

       if (dynamoDbClient == null) {
           throw new IllegalArgumentException("DynamoDB client cannot be null");
       }
       if (tableName == null || tableName.trim().isEmpty()) {
           throw new IllegalArgumentException("Table name cannot be null or
empty");
       }

       try {
           LOGGER.info("Describing MRSC global table: " + tableName);

           DescribeTableRequest request =
               DescribeTableRequest.builder().tableName(tableName).build();

           DescribeTableResponse response =
dynamoDbClient.describeTable(request);

           LOGGER.info("Table status: " + response.table().tableStatus());
           LOGGER.info("Multi-region consistency: " +
response.table().multiRegionConsistency());

           if (response.table().replicas() != null
               && !response.table().replicas().isEmpty()) {
               LOGGER.info("Number of replicas: " +
response.table().replicas().size());
               response.table()
                   .replicas()
                   .forEach(replica -> LOGGER.info(
                       "Replica region: " + replica.regionName() + ", Status: "
+ replica.replicaStatus()));
           }

           if (response.table().globalTableWitnesses() != null
```

```
                && !response.table().globalTableWitnesses().isEmpty()) {
                LOGGER.info("Number of witnesses: "
                    + response.table().globalTableWitnesses().size());
                response.table()
                    .globalTableWitnesses()
                    .forEach(witness -> LOGGER.info(
                        "Witness region: " + witness.regionName() + ", Status: "
+ witness.witnessStatus()));
            }

            return response;

        } catch (ResourceNotFoundException e) {
            LOGGER.severe("Table not found: " + tableName + " - " +
e.getMessage());
            throw DynamoDbException.builder()
                .message("Table not found: " + tableName)
                .cause(e)
                .build();
        } catch (DynamoDbException e) {
            LOGGER.severe("Failed to describe table: " + tableName + " - " +
e.getMessage());
            throw DynamoDbException.builder()
                .message("Failed to describe table: " + tableName)
                .cause(e)
                .build();
        }
    }
```

Add test items to verify MRSC strong consistency using AWS SDK for Java 2.x.

```
    public static PutItemResponse putTestItem(
        final DynamoDbClient dynamoDbClient,
        final String tableName,
        final String artist,
        final String songTitle,
        final String album,
        final String year) {

        if (dynamoDbClient == null) {
            throw new IllegalArgumentException("DynamoDB client cannot be null");
        }
```

```
        if (tableName == null || tableName.trim().isEmpty()) {
            throw new IllegalArgumentException("Table name cannot be null or
empty");
        }
        if (artist == null || artist.trim().isEmpty()) {
            throw new IllegalArgumentException("Artist cannot be null or empty");
        }
        if (songTitle == null || songTitle.trim().isEmpty()) {
            throw new IllegalArgumentException("Song title cannot be null or
empty");
        }

        try {
            LOGGER.info("Adding test item to MRSC global table: " + tableName);

            Map<String, AttributeValue> item = new HashMap<>();
            item.put("Artist", AttributeValue.builder().s(artist).build());
            item.put("SongTitle", AttributeValue.builder().s(songTitle).build());

            if (album != null && !album.trim().isEmpty()) {
                item.put("Album", AttributeValue.builder().s(album).build());
            }
            if (year != null && !year.trim().isEmpty()) {
                item.put("Year", AttributeValue.builder().n(year).build());
            }

            PutItemRequest putItemRequest =
                PutItemRequest.builder().tableName(tableName).item(item).build();

            PutItemResponse response = dynamoDbClient.putItem(putItemRequest);
            LOGGER.info("Test item added successfully with strong consistency");

            return response;

        } catch (DynamoDbException e) {
            LOGGER.severe("Failed to add test item to table: " + tableName + " -
" + e.getMessage());
            throw DynamoDbException.builder()
                .message("Failed to add test item to table: " + tableName)
                .cause(e)
                .build();
        }
    }
```

Read items with consistent reads from MRSC replicas using AWS SDK for Java 2.x.

```
   public static GetItemResponse getItemWithConsistentRead(
       final DynamoDbClient dynamoDbClient, final String tableName, final String
artist, final String songTitle) {

       if (dynamoDbClient == null) {
           throw new IllegalArgumentException("DynamoDB client cannot be null");
       }
       if (tableName == null || tableName.trim().isEmpty()) {
           throw new IllegalArgumentException("Table name cannot be null or
empty");
       }
       if (artist == null || artist.trim().isEmpty()) {
           throw new IllegalArgumentException("Artist cannot be null or empty");
       }
       if (songTitle == null || songTitle.trim().isEmpty()) {
           throw new IllegalArgumentException("Song title cannot be null or
empty");
       }

       try {
           LOGGER.info("Reading item from MRSC global table with consistent
read: " + tableName);

           Map<String, AttributeValue> key = new HashMap<>();
           key.put("Artist", AttributeValue.builder().s(artist).build());
           key.put("SongTitle", AttributeValue.builder().s(songTitle).build());

           GetItemRequest getItemRequest = GetItemRequest.builder()
               .tableName(tableName)
               .key(key)
               .consistentRead(true)
               .build();

           GetItemResponse response = dynamoDbClient.getItem(getItemRequest);

           if (response.hasItem()) {
               LOGGER.info("Item found with strong consistency - no wait time
needed");
           } else {
```

```
                LOGGER.info("Item not found");
            }

            return response;

        } catch (DynamoDbException e) {
            LOGGER.severe("Failed to read item from table: " + tableName + " - "
+ e.getMessage());
            throw DynamoDbException.builder()
                .message("Failed to read item from table: " + tableName)
                .cause(e)
                .build();
        }
    }
```

Perform conditional updates with MRSC guarantees using AWS SDK for Java 2.x.

```
    public static UpdateItemResponse performConditionalUpdate(
        final DynamoDbClient dynamoDbClient,
        final String tableName,
        final String artist,
        final String songTitle,
        final String rating) {

        if (dynamoDbClient == null) {
            throw new IllegalArgumentException("DynamoDB client cannot be null");
        }
        if (tableName == null || tableName.trim().isEmpty()) {
            throw new IllegalArgumentException("Table name cannot be null or
empty");
        }
        if (artist == null || artist.trim().isEmpty()) {
            throw new IllegalArgumentException("Artist cannot be null or empty");
        }
        if (songTitle == null || songTitle.trim().isEmpty()) {
            throw new IllegalArgumentException("Song title cannot be null or
empty");
        }
        if (rating == null || rating.trim().isEmpty()) {
            throw new IllegalArgumentException("Rating cannot be null or empty");
        }
```

```
        try {
            LOGGER.info("Performing conditional update on MRSC global table: " +
tableName);

            Map<String, AttributeValue> key = new HashMap<>();
            key.put("Artist", AttributeValue.builder().s(artist).build());
            key.put("SongTitle", AttributeValue.builder().s(songTitle).build());

            Map<String, String> expressionAttributeNames = new HashMap<>();
            expressionAttributeNames.put("#rating", "Rating");

            Map<String, AttributeValue> expressionAttributeValues = new
HashMap<>();
            expressionAttributeValues.put(
                ":rating", AttributeValue.builder().n(rating).build());

            UpdateItemRequest updateItemRequest = UpdateItemRequest.builder()
                .tableName(tableName)
                .key(key)
                .updateExpression("SET #rating = :rating")
                .conditionExpression("attribute_exists(Artist)")
                .expressionAttributeNames(expressionAttributeNames)
                .expressionAttributeValues(expressionAttributeValues)
                .build();

            UpdateItemResponse response =
dynamoDbClient.updateItem(updateItemRequest);
            LOGGER.info("Conditional update successful - demonstrates strong
consistency");

            return response;

        } catch (ConditionalCheckFailedException e) {
            LOGGER.warning("Conditional check failed: " + e.getMessage());
            throw e;
        } catch (DynamoDbException e) {
            LOGGER.severe("Failed to perform conditional update: " + tableName +
" - " + e.getMessage());
            throw DynamoDbException.builder()
                .message("Failed to perform conditional update: " + tableName)
                .cause(e)
                .build();
        }
    }
```

Wait for MRSC replicas and witnesses to become active using AWS SDK for Java 2.x.

```
   public static void waitForMRSCReplicasActive(
       final DynamoDbClient dynamoDbClient, final String tableName, final int
maxWaitTimeSeconds)
       throws InterruptedException {

       if (dynamoDbClient == null) {
           throw new IllegalArgumentException("DynamoDB client cannot be null");
       }
       if (tableName == null || tableName.trim().isEmpty()) {
           throw new IllegalArgumentException("Table name cannot be null or
empty");
       }
       if (maxWaitTimeSeconds <= 0) {
           throw new IllegalArgumentException("Max wait time must be positive");
       }

       try {
           LOGGER.info("Waiting for MRSC replicas and witnesses to become
active: " + tableName);

           final long startTime = System.currentTimeMillis();
           final long maxWaitTimeMillis = maxWaitTimeSeconds * 1000L;
           int backoffSeconds = 5; // Start with 5 second intervals
           final int maxBackoffSeconds = 30; // Cap at 30 seconds

           while (System.currentTimeMillis() - startTime < maxWaitTimeMillis) {
               DescribeTableResponse response =
describeMRSCTable(dynamoDbClient, tableName);

               boolean allActive = true;
               StringBuilder statusReport = new StringBuilder();

               if (response.table().multiRegionConsistency() == null
                   || !MultiRegionConsistency.STRONG
                       .toString()

.equals(response.table().multiRegionConsistency().toString())) {
                   allActive = false;
                   statusReport
```

```
                        .append("MultiRegionConsistency: ")
                        .append(response.table().multiRegionConsistency())
                        .append(" ");
                }
                if (response.table().replicas() == null
                    || response.table().replicas().isEmpty()) {
                    allActive = false;
                    statusReport.append("No replicas found. ");
                }
                if (response.table().globalTableWitnesses() == null
                    || response.table().globalTableWitnesses().isEmpty()) {
                    allActive = false;
                    statusReport.append("No witnesses found. ");
                }

                // Check table status
                if (!"ACTIVE".equals(response.table().tableStatus().toString()))
{
                    allActive = false;
                    statusReport
                        .append("Table: ")
                        .append(response.table().tableStatus())
                        .append(" ");
                }

                // Check replica status
                if (response.table().replicas() != null) {
                    for (var replica : response.table().replicas()) {
                        if (!"ACTIVE".equals(replica.replicaStatus().toString()))
{
                            allActive = false;
                            statusReport
                                .append("Replica(")
                                .append(replica.regionName())
                                .append("): ")
                                .append(replica.replicaStatus())
                                .append(" ");
                        }
                    }
                }

                // Check witness status
                if (response.table().globalTableWitnesses() != null) {
                    for (var witness : response.table().globalTableWitnesses()) {
```

```
                        if (!"ACTIVE".equals(witness.witnessStatus().toString()))
{
                            allActive = false;
                            statusReport
                                .append("Witness(")
                                .append(witness.regionName())
                                .append("): ")
                                .append(witness.witnessStatus())
                                .append(" ");
                        }
                    }
                }

                if (allActive) {
                    LOGGER.info("All MRSC replicas and witnesses are now active:
" + tableName);
                    return;
                }

                LOGGER.info("Waiting for MRSC components to become active.
Status: " + statusReport.toString());
                LOGGER.info("Next check in " + backoffSeconds + " seconds...");

                tempWait(backoffSeconds);

                // Exponential backoff with cap
                backoffSeconds = Math.min(backoffSeconds * 2, maxBackoffSeconds);
            }

            throw DynamoDbException.builder()
                .message("Timeout waiting for MRSC replicas to become active
after " + maxWaitTimeSeconds + " seconds")
                .build();

        } catch (DynamoDbException | InterruptedException e) {
            LOGGER.severe("Failed to wait for MRSC replicas to become active: " +
tableName + " - " + e.getMessage());
            throw e;
        }
    }
```

Clean up MRSC replicas and witnesses using AWS SDK for Java 2.x.

```
   public static UpdateTableResponse cleanupMRSCReplicas(
       final DynamoDbClient dynamoDbClient,
       final String tableName,
       final Region replicaRegion,
       final Region witnessRegion) {

       if (dynamoDbClient == null) {
           throw new IllegalArgumentException("DynamoDB client cannot be null");
       }
       if (tableName == null || tableName.trim().isEmpty()) {
           throw new IllegalArgumentException("Table name cannot be null or
empty");
       }
       if (replicaRegion == null) {
           throw new IllegalArgumentException("Replica region cannot be null");
       }
       if (witnessRegion == null) {
           throw new IllegalArgumentException("Witness region cannot be null");
       }

       try {
           LOGGER.info("Cleaning up MRSC replicas and witnesses for table: " +
tableName);

           // Remove replica using ReplicationGroupUpdate
           ReplicationGroupUpdate replicaUpdate =
ReplicationGroupUpdate.builder()
               .delete(DeleteReplicationGroupMemberAction.builder()
                   .regionName(replicaRegion.id())
                   .build())
               .build();

           // Remove witness
           GlobalTableWitnessGroupUpdate witnessUpdate =
GlobalTableWitnessGroupUpdate.builder()
               .delete(DeleteGlobalTableWitnessGroupMemberAction.builder()
                   .regionName(witnessRegion.id())
                   .build())
               .build();

           UpdateTableRequest updateTableRequest = UpdateTableRequest.builder()
               .tableName(tableName)
               .replicaUpdates(List.of(replicaUpdate))
```

```
                .globalTableWitnessUpdates(List.of(witnessUpdate))
                .build();

            UpdateTableResponse response =
dynamoDbClient.updateTable(updateTableRequest);
            LOGGER.info("MRSC cleanup initiated - removing replica and witness.
Response: " + response);

            return response;

        } catch (DynamoDbException e) {
            LOGGER.severe("Failed to cleanup MRSC replicas: " + tableName + " - "
+ e.getMessage());
            throw DynamoDbException.builder()
                .message("Failed to cleanup MRSC replicas: " + tableName)
                .cause(e)
                .build();
        }
    }
```

Complete MRSC workflow demonstration using AWS SDK for Java 2.x.

```
    public static void demonstrateCompleteMRSCWorkflow(
        final DynamoDbClient primaryClient,
        final DynamoDbClient replicaClient,
        final String tableName,
        final Region replicaRegion,
        final Region witnessRegion)
        throws InterruptedException {

        if (primaryClient == null) {
            throw new IllegalArgumentException("Primary DynamoDB client cannot be
null");
        }
        if (replicaClient == null) {
            throw new IllegalArgumentException("Replica DynamoDB client cannot be
null");
        }
        if (tableName == null || tableName.trim().isEmpty()) {
            throw new IllegalArgumentException("Table name cannot be null or
empty");
        }
```

```
        if (replicaRegion == null) {
            throw new IllegalArgumentException("Replica region cannot be null");
        }
        if (witnessRegion == null) {
            throw new IllegalArgumentException("Witness region cannot be null");
        }

        try {
            LOGGER.info("=== Starting Complete MRSC Workflow Demonstration ===");

            // Step 1: Create an empty single-Region table
            LOGGER.info("Step 1: Creating empty single-Region table");
            createRegionalTable(primaryClient, tableName);

            // Use the existing GlobalTableOperations method for basic table
waiting
            LOGGER.info("Intermediate step: Waiting for table [" + tableName + "]
to become active before continuing");
            GlobalTableOperations.waitForTableActive(primaryClient, tableName);

            // Step 2: Convert to MRSC with replica and witness
            LOGGER.info("Step 2: Converting to MRSC with replica and witness");
            convertToMRSCWithWitness(primaryClient, tableName, replicaRegion,
witnessRegion);

            // Wait for MRSC conversion to complete using MRSC-specific waiter
            LOGGER.info("Waiting for MRSC conversion to complete...");
            waitForMRSCReplicasActive(primaryClient, tableName);

            LOGGER.info("Intermediate step: Waiting for table [" + tableName + "]
to become active before continuing");
            GlobalTableOperations.waitForTableActive(primaryClient, tableName);

            // Step 3: Verify MRSC configuration
            LOGGER.info("Step 3: Verifying MRSC configuration");
            describeMRSCTable(primaryClient, tableName);

            // Step 4: Test strong consistency with data operations
            LOGGER.info("Step 4: Testing strong consistency with data
operations");

            // Add test item to primary region
            putTestItem(primaryClient, tableName, "The Beatles", "Hey Jude", "The
Beatles 1967-1970", "1968");
```

```
            // Immediately read from replica region (no wait needed with MRSC)
            LOGGER.info("Reading from replica region immediately (strong
consistency):");
            GetItemResponse getResponse =
                getItemWithConsistentRead(replicaClient, tableName, "The
Beatles", "Hey Jude");

            if (getResponse.hasItem()) {
                LOGGER.info("# Strong consistency verified - item immediately
available in replica region");
            } else {
                LOGGER.warning("# Item not found in replica region");
            }

            // Test conditional update from replica region
            LOGGER.info("Testing conditional update from replica region:");
            performConditionalUpdate(replicaClient, tableName, "The Beatles",
"Hey Jude", "5");
            LOGGER.info("# Conditional update successful - demonstrates strong
consistency");

            // Step 5: Cleanup
            LOGGER.info("Step 5: Cleaning up resources");
            cleanupMRSCReplicas(primaryClient, tableName, replicaRegion,
witnessRegion);

            // Wait for cleanup to complete using basic table waiter
            LOGGER.info("Waiting for replica cleanup to complete...");
            GlobalTableOperations.waitForTableActive(primaryClient, tableName);

            // "Halt" until replica/witness cleanup is complete
            DescribeTableResponse cleanupVerification =
describeMRSCTable(primaryClient, tableName);
            int backoffSeconds = 5; // Start with 5 second intervals
            while (cleanupVerification.table().multiRegionConsistency() != null)
{
                LOGGER.info("Waiting additional time (" + backoffSeconds + "
seconds) for MRSC cleanup to complete...");
                tempWait(backoffSeconds);

                // Exponential backoff with cap
                backoffSeconds = Math.min(backoffSeconds * 2, 30);
```

```
                cleanupVerification = describeMRSCTable(primaryClient,
tableName);
            }

            // Delete the primary table
            deleteTable(primaryClient, tableName);

            LOGGER.info("=== MRSC Workflow Demonstration Complete ===");
            LOGGER.info("");
            LOGGER.info("Key benefits of Multi-Region Strong Consistency
(MRSC):");
            LOGGER.info("- Immediate consistency across all regions (no eventual
consistency delays)");
            LOGGER.info("- Simplified application logic (no need to handle
eventual consistency)");
            LOGGER.info("- Support for conditional writes and transactions across
regions");
            LOGGER.info("- Consistent read operations from any region without
waiting");

        } catch (DynamoDbException | InterruptedException e) {
            LOGGER.severe("MRSC workflow failed: " + e.getMessage());
            throw e;
        }
    }
```

- For API details, see the following topics in *AWS SDK for Java 2.x API Reference*.
  - CreateTable
  - DeleteTable
  - DescribeTable
  - GetItem
  - PutItem
  - UpdateItem
  - UpdateTable

For a complete list of AWS SDK developer guides and code examples, see Using DynamoDB with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Create and manage DynamoDB global tables demonstrating MREC using an AWS SDK

The following code example shows how to create and manage DynamoDB global tables with replicas across multiple Regions.

- Create a table with Global Secondary Index and DynamoDB Streams.
- Add replicas in different Regions to create a global table.
- Remove replicas from a global table.
- Add test items to verify replication across Regions.
- Describe global table configuration and replica status.

Java

SDK for Java 2.x

Create a table with Global Secondary Index and DynamoDB Streams using AWS SDK for Java 2.x.

```
   public static CreateTableResponse createTableWithGSI(
       final DynamoDbClient dynamoDbClient, final String tableName, final String
indexName) {

       if (dynamoDbClient == null) {
           throw new IllegalArgumentException("DynamoDB client cannot be null");
       }
       if (tableName == null || tableName.trim().isEmpty()) {
           throw new IllegalArgumentException("Table name cannot be null or
empty");
       }
       if (indexName == null || indexName.trim().isEmpty()) {
           throw new IllegalArgumentException("Index name cannot be null or
empty");
       }

       try {
           LOGGER.info("Creating table: " + tableName + " with GSI: " +
indexName);

           CreateTableRequest createTableRequest = CreateTableRequest.builder()
```

```
                .tableName(tableName)
                .attributeDefinitions(
                    AttributeDefinition.builder()
                        .attributeName("Artist")
                        .attributeType(ScalarAttributeType.S)
                        .build(),
                    AttributeDefinition.builder()
                        .attributeName("SongTitle")
                        .attributeType(ScalarAttributeType.S)
                        .build())
                .keySchema(
                    KeySchemaElement.builder()
                        .attributeName("Artist")
                        .keyType(KeyType.HASH)
                        .build(),
                    KeySchemaElement.builder()
                        .attributeName("SongTitle")
                        .keyType(KeyType.RANGE)
                        .build())
                .billingMode(BillingMode.PAY_PER_REQUEST)
                .globalSecondaryIndexes(GlobalSecondaryIndex.builder()
                    .indexName(indexName)
                    .keySchema(KeySchemaElement.builder()
                        .attributeName("SongTitle")
                        .keyType(KeyType.HASH)
                        .build())
                    .projection(

Projection.builder().projectionType(ProjectionType.ALL).build())
                    .build())
                .streamSpecification(StreamSpecification.builder()
                    .streamEnabled(true)
                    .streamViewType(StreamViewType.NEW_AND_OLD_IMAGES)
                    .build())
                .build();

            CreateTableResponse response =
dynamoDbClient.createTable(createTableRequest);
            LOGGER.info("Table creation initiated. Status: "
                + response.tableDescription().tableStatus());

            return response;

        } catch (DynamoDbException e) {
```

```
            LOGGER.severe("Failed to create table: " + tableName + " - " +
e.getMessage());
            throw e;
        }
    }
```

Wait for a table to become active using AWS SDK for Java 2.x.

```
    public static void waitForTableActive(final DynamoDbClient dynamoDbClient,
final String tableName) {

        if (dynamoDbClient == null) {
            throw new IllegalArgumentException("DynamoDB client cannot be null");
        }
        if (tableName == null || tableName.trim().isEmpty()) {
            throw new IllegalArgumentException("Table name cannot be null or
empty");
        }

        try {
            LOGGER.info("Waiting for table to become active: " + tableName);

            try (DynamoDbWaiter waiter =
                DynamoDbWaiter.builder().client(dynamoDbClient).build()) {
                DescribeTableRequest request =
                    DescribeTableRequest.builder().tableName(tableName).build();

                waiter.waitUntilTableExists(request);
                LOGGER.info("Table is now active: " + tableName);
            }

        } catch (DynamoDbException e) {
            LOGGER.severe("Failed to wait for table to become active: " +
tableName + " - " + e.getMessage());
            throw e;
        }
    }
```

Add a replica to create or extend a global table using AWS SDK for Java 2.x.

```
    public static UpdateTableResponse addReplica(
```

```
        final DynamoDbClient dynamoDbClient,
        final String tableName,
        final Region replicaRegion,
        final String indexName,
        final Long readCapacity) {

        if (dynamoDbClient == null) {
            throw new IllegalArgumentException("DynamoDB client cannot be null");
        }
        if (tableName == null || tableName.trim().isEmpty()) {
            throw new IllegalArgumentException("Table name cannot be null or
empty");
        }
        if (replicaRegion == null) {
            throw new IllegalArgumentException("Replica region cannot be null");
        }
        if (indexName == null || indexName.trim().isEmpty()) {
            throw new IllegalArgumentException("Index name cannot be null or
empty");
        }
        if (readCapacity == null || readCapacity <= 0) {
            throw new IllegalArgumentException("Read capacity must be a positive
number");
        }

        try {
            LOGGER.info("Adding replica in region: " + replicaRegion.id() + " for
table: " + tableName);

            // Create a ReplicationGroupUpdate for adding a replica
            ReplicationGroupUpdate replicationGroupUpdate =
ReplicationGroupUpdate.builder()
                .create(builder -> builder.regionName(replicaRegion.id())
                    .globalSecondaryIndexes(ReplicaGlobalSecondaryIndex.builder()
                        .indexName(indexName)

.provisionedThroughputOverride(ProvisionedThroughputOverride.builder()
                            .readCapacityUnits(readCapacity)
                            .build())
                        .build())
                    .build())
                .build();

            UpdateTableRequest updateTableRequest = UpdateTableRequest.builder()
```

```
                .tableName(tableName)
                .replicaUpdates(replicationGroupUpdate)
                .build();

            UpdateTableResponse response =
dynamoDbClient.updateTable(updateTableRequest);
            LOGGER.info("Replica addition initiated in region: " +
replicaRegion.id());

            return response;

        } catch (DynamoDbException e) {
            LOGGER.severe("Failed to add replica in region: " +
replicaRegion.id() + " - " + e.getMessage());
            throw e;
        }
    }
```

Remove a replica from a global table using AWS SDK for Java 2.x.

```
    public static UpdateTableResponse removeReplica(
        final DynamoDbClient dynamoDbClient, final String tableName, final Region
replicaRegion) {

        if (dynamoDbClient == null) {
            throw new IllegalArgumentException("DynamoDB client cannot be null");
        }
        if (tableName == null || tableName.trim().isEmpty()) {
            throw new IllegalArgumentException("Table name cannot be null or
empty");
        }
        if (replicaRegion == null) {
            throw new IllegalArgumentException("Replica region cannot be null");
        }

        try {
            LOGGER.info("Removing replica in region: " + replicaRegion.id() + "
for table: " + tableName);

            // Create a ReplicationGroupUpdate for removing a replica
            ReplicationGroupUpdate replicationGroupUpdate =
ReplicationGroupUpdate.builder()
```

```
                .delete(builder ->
builder.regionName(replicaRegion.id()).build())
                .build();

            UpdateTableRequest updateTableRequest = UpdateTableRequest.builder()
                .tableName(tableName)
                .replicaUpdates(replicationGroupUpdate)
                .build();

            UpdateTableResponse response =
dynamoDbClient.updateTable(updateTableRequest);
            LOGGER.info("Replica removal initiated in region: " +
replicaRegion.id());

            return response;

        } catch (DynamoDbException e) {
            LOGGER.severe("Failed to remove replica in region: " +
replicaRegion.id() + " - " + e.getMessage());
            throw e;
        }
    }
```

Add test items to verify replication using AWS SDK for Java 2.x.

```
    public static PutItemResponse putTestItem(
        final DynamoDbClient dynamoDbClient, final String tableName, final String
artist, final String songTitle) {

        if (dynamoDbClient == null) {
            throw new IllegalArgumentException("DynamoDB client cannot be null");
        }
        if (tableName == null || tableName.trim().isEmpty()) {
            throw new IllegalArgumentException("Table name cannot be null or
empty");
        }
        if (artist == null || artist.trim().isEmpty()) {
            throw new IllegalArgumentException("Artist cannot be null or empty");
        }
        if (songTitle == null || songTitle.trim().isEmpty()) {
            throw new IllegalArgumentException("Song title cannot be null or
empty");
```

```
        }

        try {
            LOGGER.info("Adding test item to table: " + tableName);

            Map<String,
software.amazon.awssdk.services.dynamodb.model.AttributeValue> item = new
HashMap<>();
            item.put(
                "Artist",

software.amazon.awssdk.services.dynamodb.model.AttributeValue.builder()
                    .s(artist)
                    .build());
            item.put(
                "SongTitle",

software.amazon.awssdk.services.dynamodb.model.AttributeValue.builder()
                    .s(songTitle)
                    .build());

            PutItemRequest putItemRequest =
                PutItemRequest.builder().tableName(tableName).item(item).build();

            PutItemResponse response = dynamoDbClient.putItem(putItemRequest);
            LOGGER.info("Test item added successfully");

            return response;

        } catch (DynamoDbException e) {
            LOGGER.severe("Failed to add test item to table: " + tableName + " -
" + e.getMessage());
            throw e;
        }
    }
```

Describe global table configuration and replicas using AWS SDK for Java 2.x.

```
    public static DescribeTableResponse describeTable(final DynamoDbClient
dynamoDbClient, final String tableName) {

        if (dynamoDbClient == null) {
```

```
            throw new IllegalArgumentException("DynamoDB client cannot be null");
        }
        if (tableName == null || tableName.trim().isEmpty()) {
            throw new IllegalArgumentException("Table name cannot be null or
empty");
        }

        try {
            LOGGER.info("Describing table: " + tableName);

            DescribeTableRequest request =
                DescribeTableRequest.builder().tableName(tableName).build();

            DescribeTableResponse response =
dynamoDbClient.describeTable(request);

            LOGGER.info("Table status: " + response.table().tableStatus());
            if (response.table().replicas() != null
                && !response.table().replicas().isEmpty()) {
                LOGGER.info("Number of replicas: " +
response.table().replicas().size());
                response.table()
                    .replicas()
                    .forEach(replica -> LOGGER.info(
                        "Replica region: " + replica.regionName() + ", Status: "
+ replica.replicaStatus()));
            }

            return response;

        } catch (ResourceNotFoundException e) {
            LOGGER.severe("Table not found: " + tableName + " - " +
e.getMessage());
            throw e;
        } catch (DynamoDbException e) {
            LOGGER.severe("Failed to describe table: " + tableName + " - " +
e.getMessage());
            throw e;
        }
    }
```

Complete example of global table operations using AWS SDK for Java 2.x.

```
   public static void exampleUsage(final Region sourceRegion, final Region
replicaRegion) {

       String tableName = "Music";
       String indexName = "SongTitleIndex";
       Long readCapacity = 15L;

       // Create DynamoDB client for the source region
       try (DynamoDbClient dynamoDbClient =
           DynamoDbClient.builder().region(sourceRegion).build()) {

           try {
               // Step 1: Create the initial table with GSI and streams
               LOGGER.info("Step 1: Creating table in source region: " +
sourceRegion.id());
               createTableWithGSI(dynamoDbClient, tableName, indexName);

               // Step 2: Wait for table to become active
               LOGGER.info("Step 2: Waiting for table to become active");
               waitForTableActive(dynamoDbClient, tableName);

               // Step 3: Add replica in destination region
               LOGGER.info("Step 3: Adding replica in region: " +
replicaRegion.id());
               addReplica(dynamoDbClient, tableName, replicaRegion, indexName,
readCapacity);

               // Step 4: Wait a moment for replica creation to start
               Thread.sleep(5000);

               // Step 5: Describe table to view replica information
               LOGGER.info("Step 5: Describing table to view replicas");
               describeTable(dynamoDbClient, tableName);

               // Step 6: Add a test item to verify replication
               LOGGER.info("Step 6: Adding test item to verify replication");
               putTestItem(dynamoDbClient, tableName, "TestArtist", "TestSong");

               LOGGER.info("Global table setup completed successfully!");
               LOGGER.info("You can verify replication by checking the item in
region: " + replicaRegion.id());

               // Step 7: Remove replica and clean up table
```

```
                LOGGER.info("Step 7: Removing replica from region: " +
replicaRegion.id());
                removeReplica(dynamoDbClient, tableName, replicaRegion);
                DeleteTableResponse deleteTableResponse =
dynamoDbClient.deleteTable(
                    DeleteTableRequest.builder().tableName(tableName).build());
                LOGGER.info("MREC global table demonstration completed
successfully!");

            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
                throw new RuntimeException("Thread was interrupted", e);
            } catch (DynamoDbException e) {
                LOGGER.severe("DynamoDB operation failed: " + e.getMessage());
                throw e;
            }
        }
    }
```

- For API details, see the following topics in *AWS SDK for Java 2.x API Reference*.
  - CreateTable
  - DescribeTable
  - PutItem
  - UpdateTable

For a complete list of AWS SDK developer guides and code examples, see Using DynamoDB with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Delete DynamoDB data using PartiQL DELETE statements with an AWS SDK

The following code example shows how to delete data using PartiQL DELETE statements.

JavaScript

SDK for JavaScript (v3)

Delete items from a DynamoDB table using PartiQL DELETE statements with AWS SDK for JavaScript.

```
/**
 * This example demonstrates how to delete items from a DynamoDB table using
 PartiQL.
 * It shows different ways to delete documents with various index types.
 */
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
import {
  DynamoDBDocumentClient,
  ExecuteStatementCommand,
  BatchExecuteStatementCommand,
} from "@aws-sdk/lib-dynamodb";

/**
 * Delete a single item by its partition key using PartiQL.
 *
 * @param tableName - The name of the DynamoDB table
 * @param partitionKeyName - The name of the partition key attribute
 * @param partitionKeyValue - The value of the partition key
 * @returns The response from the ExecuteStatementCommand
 */
export const deleteItemByPartitionKey = async (
  tableName: string,
  partitionKeyName: string,
  partitionKeyValue: string | number
) => {
  const client = new DynamoDBClient({});
  const docClient = DynamoDBDocumentClient.from(client);

  const params = {
    Statement: `DELETE FROM "${tableName}" WHERE ${partitionKeyName} = ?`,
    Parameters: [partitionKeyValue],
  };

  try {
    const data = await docClient.send(new ExecuteStatementCommand(params));
    console.log("Item deleted successfully");
```

```
    return data;
  } catch (err) {
    console.error("Error deleting item:", err);
    throw err;
  }
};

/**
 * Delete an item by its composite key (partition key + sort key) using PartiQL.
 *
 * @param tableName - The name of the DynamoDB table
 * @param partitionKeyName - The name of the partition key attribute
 * @param partitionKeyValue - The value of the partition key
 * @param sortKeyName - The name of the sort key attribute
 * @param sortKeyValue - The value of the sort key
 * @returns The response from the ExecuteStatementCommand
 */
export const deleteItemByCompositeKey = async (
  tableName: string,
  partitionKeyName: string,
  partitionKeyValue: string | number,
  sortKeyName: string,
  sortKeyValue: string | number
) => {
  const client = new DynamoDBClient({});
  const docClient = DynamoDBDocumentClient.from(client);

  const params = {
    Statement: `DELETE FROM "${tableName}" WHERE ${partitionKeyName} = ? AND
 ${sortKeyName} = ?`,
    Parameters: [partitionKeyValue, sortKeyValue],
  };

  try {
    const data = await docClient.send(new ExecuteStatementCommand(params));
    console.log("Item deleted successfully");
    return data;
  } catch (err) {
    console.error("Error deleting item:", err);
    throw err;
  }
};

/**
```

```
 * Delete an item with a condition to ensure the delete only happens if a
 condition is met.
 *
 * @param tableName - The name of the DynamoDB table
 * @param partitionKeyName - The name of the partition key attribute
 * @param partitionKeyValue - The value of the partition key
 * @param conditionAttribute - The attribute to check in the condition
 * @param conditionValue - The value to compare against in the condition
 * @returns The response from the ExecuteStatementCommand
 */
export const deleteItemWithCondition = async (
  tableName: string,
  partitionKeyName: string,
  partitionKeyValue: string | number,
  conditionAttribute: string,
  conditionValue: any
) => {
  const client = new DynamoDBClient({});
  const docClient = DynamoDBDocumentClient.from(client);

  const params = {
    Statement: `DELETE FROM "${tableName}" WHERE ${partitionKeyName} = ? AND
 ${conditionAttribute} = ?`,
    Parameters: [partitionKeyValue, conditionValue],
  };

  try {
    const data = await docClient.send(new ExecuteStatementCommand(params));
    console.log("Item deleted with condition successfully");
    return data;
  } catch (err) {
    console.error("Error deleting item with condition:", err);
    throw err;
  }
};

/**
 * Batch delete multiple items using PartiQL.
 *
 * @param tableName - The name of the DynamoDB table
 * @param keys - Array of objects containing key information
 * @returns The response from the BatchExecuteStatementCommand
 */
export const batchDeleteItems = async (
```

```
  tableName: string,
  keys: Array<{
    partitionKeyName: string;
    partitionKeyValue: string | number;
    sortKeyName?: string;
    sortKeyValue?: string | number;
  }>
) => {
  const client = new DynamoDBClient({});
  const docClient = DynamoDBDocumentClient.from(client);

  // Create statements for each delete
  const statements = keys.map((key) => {
    if (key.sortKeyName && key.sortKeyValue !== undefined) {
      return {
        Statement: `DELETE FROM "${tableName}" WHERE ${key.partitionKeyName} = ?
 AND ${key.sortKeyName} = ?`,
        Parameters: [key.partitionKeyValue, key.sortKeyValue],
      };
    } else {
      return {
        Statement: `DELETE FROM "${tableName}" WHERE ${key.partitionKeyName} = ?
`,
        Parameters: [key.partitionKeyValue],
      };
    }
  });

  const params = {
    Statements: statements,
  };

  try {
    const data = await docClient.send(new BatchExecuteStatementCommand(params));
    console.log("Items batch deleted successfully");
    return data;
  } catch (err) {
    console.error("Error batch deleting items:", err);
    throw err;
  }
};

/**
 * Delete multiple items that match a filter condition.
```

```
 * Note: This performs a scan operation which can be expensive on large tables.
 *
 * @param tableName - The name of the DynamoDB table
 * @param filterAttribute - The attribute to filter on
 * @param filterValue - The value to filter by
 * @returns The response from the ExecuteStatementCommand
 */
export const deleteItemsByFilter = async (
  tableName: string,
  filterAttribute: string,
  filterValue: any
) => {
  const client = new DynamoDBClient({});
  const docClient = DynamoDBDocumentClient.from(client);

  const params = {
    Statement: `DELETE FROM "${tableName}" WHERE ${filterAttribute} = ?`,
    Parameters: [filterValue],
  };

  try {
    const data = await docClient.send(new ExecuteStatementCommand(params));
    console.log("Items deleted by filter successfully");
    return data;
  } catch (err) {
    console.error("Error deleting items by filter:", err);
    throw err;
  }
};

/**
 * Example usage showing how to delete items with different index types
 */
export const deleteExamples = async () => {
  // Delete an item by partition key (simple primary key)
  await deleteItemByPartitionKey("UsersTable", "userId", "user123");

  // Delete an item by composite key (partition key + sort key)
  await deleteItemByCompositeKey(
    "OrdersTable",
    "orderId",
    "order456",
    "productId",
    "prod789"
```

```
  );

  // Delete with a condition
  await deleteItemWithCondition(
    "UsersTable",
    "userId",
    "user789",
    "userStatus",
    "inactive"
  );

  // Batch delete multiple items
  await batchDeleteItems("UsersTable", [
    { partitionKeyName: "userId", partitionKeyValue: "user234" },
    { partitionKeyName: "userId", partitionKeyValue: "user345" },
  ]);

  // Batch delete items with composite keys
  await batchDeleteItems("OrdersTable", [
    {
      partitionKeyName: "orderId",
      partitionKeyValue: "order567",
      sortKeyName: "productId",
      sortKeyValue: "prod123",
    },
    {
      partitionKeyName: "orderId",
      partitionKeyValue: "order678",
      sortKeyName: "productId",
      sortKeyValue: "prod456",
    },
  ]);

  // Delete items by filter (use with caution)
  await deleteItemsByFilter("UsersTable", "userStatus", "deleted");
};
```

- For API details, see the following topics in *AWS SDK for JavaScript API Reference*.
  - BatchExecuteStatement
  - ExecuteStatement

For a complete list of AWS SDK developer guides and code examples, see Using DynamoDB with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Detect PPE in images with Amazon Rekognition using an AWS SDK

The following code example shows how to build an app that uses Amazon Rekognition to detect Personal Protective Equipment (PPE) in images.

Java

SDK for Java 2.x

Shows how to create an AWS Lambda function that detects images with Personal Protective Equipment.

For complete source code and instructions on how to set up and run, see the full example on GitHub.

Services used in this example

- DynamoDB
- Amazon Rekognition
- Amazon S3
- Amazon SES

For a complete list of AWS SDK developer guides and code examples, see Using DynamoDB with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Insert DynamoDB data using PartiQL INSERT statements with an AWS SDK

The following code example shows how to insert data using PartiQL INSERT statements.

JavaScript

SDK for JavaScript (v3)

Insert items into a DynamoDB table using PartiQL INSERT statements with AWS SDK for JavaScript.

```
/**
 * This example demonstrates how to insert items into a DynamoDB table using
 PartiQL.
 * It shows different ways to insert documents with various index types.
 */
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
import {
  DynamoDBDocumentClient,
  ExecuteStatementCommand,
  BatchExecuteStatementCommand,
} from "@aws-sdk/lib-dynamodb";

/**
 * Insert a single item into a DynamoDB table using PartiQL.
 *
 * @param tableName - The name of the DynamoDB table
 * @param item - The item to insert
 * @returns The response from the ExecuteStatementCommand
 */
export const insertItem = async (tableName: string, item: Record<string, any>) =>
 {
  const client = new DynamoDBClient({});
  const docClient = DynamoDBDocumentClient.from(client);

  // Convert the item to a string representation for PartiQL
  const itemString = JSON.stringify(item).replace(/"([^"]+)":/g, '$1:');

  const params = {
    Statement: `INSERT INTO "${tableName}" VALUE ${itemString}`,
  };

  try {
    const data = await docClient.send(new ExecuteStatementCommand(params));
    console.log("Item inserted successfully");
    return data;
  } catch (err) {
```

```
    console.error("Error inserting item:", err);
    throw err;
  }
};

/**
 * Insert multiple items into a DynamoDB table using PartiQL batch operation.
 * This is more efficient than inserting items one by one.
 *
 * @param tableName - The name of the DynamoDB table
 * @param items - Array of items to insert
 * @returns The response from the BatchExecuteStatementCommand
 */
export const batchInsertItems = async (tableName: string, items: Record<string,
 any>[]) => {
  const client = new DynamoDBClient({});
  const docClient = DynamoDBDocumentClient.from(client);

  // Create statements for each item
  const statements = items.map((item) => {
    const itemString = JSON.stringify(item).replace(/"([^"]+)":/g, '$1:');
    return {
      Statement: `INSERT INTO "${tableName}" VALUE ${itemString}`,
    };
  });

  const params = {
    Statements: statements,
  };

  try {
    const data = await docClient.send(new BatchExecuteStatementCommand(params));
    console.log("Items inserted successfully");
    return data;
  } catch (err) {
    console.error("Error batch inserting items:", err);
    throw err;
  }
};

/**
 * Insert an item with a condition to prevent overwriting existing items.
 * This is useful for ensuring you don't accidentally overwrite data.
 *
```

```
 * @param tableName - The name of the DynamoDB table
 * @param item - The item to insert
 * @param partitionKeyName - The name of the partition key attribute
 * @returns The response from the ExecuteStatementCommand
 */
export const insertItemWithCondition = async (
  tableName: string,
  item: Record<string, any>,
  partitionKeyName: string
) => {
  const client = new DynamoDBClient({});
  const docClient = DynamoDBDocumentClient.from(client);

  const itemString = JSON.stringify(item).replace(/"([^"]+)":/g, '$1:');
  const partitionKeyValue = JSON.stringify(item[partitionKeyName]);

  const params = {
    Statement: `INSERT INTO "${tableName}" VALUE ${itemString} WHERE
 attribute_not_exists(${partitionKeyName})`,
    Parameters: [{ S: partitionKeyValue }],
  };

  try {
    const data = await docClient.send(new ExecuteStatementCommand(params));
    console.log("Item inserted with condition successfully");
    return data;
  } catch (err) {
    console.error("Error inserting item with condition:", err);
    throw err;
  }
};

/**
 * Example usage showing how to insert items with different index types
 */
export const insertExamples = async () => {
  // Example table with a simple primary key (just partition key)
  const simpleKeyItem = {
    userId: "user123",
    name: "John Doe",
    email: "john@example.com",
  };
  await insertItem("UsersTable", simpleKeyItem);
```

```
// Example table with composite key (partition key + sort key)
const compositeKeyItem = {
  orderId: "order456",
  productId: "prod789",
  quantity: 2,
  price: 29.99,
};
await insertItem("OrdersTable", compositeKeyItem);

// Example with Global Secondary Index (GSI)
// The GSI might be on the email attribute
const gsiItem = {
  userId: "user789",
  email: "jane@example.com",
  name: "Jane Smith",
  userType: "premium",  // This could be part of a GSI
};
await insertItem("UsersTable", gsiItem);

// Example with Local Secondary Index (LSI)
// LSI uses the same partition key but different sort key
const lsiItem = {
  orderId: "order567",  // Partition key
  productId: "prod123", // Sort key for the table
  orderDate: "2023-11-15", // Potential sort key for an LSI
  quantity: 1,
  price: 19.99,
};
await insertItem("OrdersTable", lsiItem);

// Batch insert example with multiple items
const batchItems = [
  {
    userId: "user234",
    name: "Alice Johnson",
    email: "alice@example.com",
  },
  {
    userId: "user345",
    name: "Bob Williams",
    email: "bob@example.com",
  },
];
await batchInsertItems("UsersTable", batchItems);
```

```
};
```

- For API details, see the following topics in *AWS SDK for JavaScript API Reference*.
  - BatchExecuteStatement
  - ExecuteStatement

For a complete list of AWS SDK developer guides and code examples, see Using DynamoDB with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Invoke a Lambda function from a browser

The following code example shows how to invoke an AWS Lambda function from a browser.

JavaScript

SDK for JavaScript (v2)

You can create a browser-based application that uses an AWS Lambda function to update an Amazon DynamoDB table with user selections.

For complete source code and instructions on how to set up and run, see the full example on GitHub.

Services used in this example

- DynamoDB
- Lambda

SDK for JavaScript (v3)

You can create a browser-based application that uses an AWS Lambda function to update an Amazon DynamoDB table with user selections. This app uses AWS SDK for JavaScript v3.

For complete source code and instructions on how to set up and run, see the full example on GitHub.

Services used in this example

- DynamoDB
- Lambda

For a complete list of AWS SDK developer guides and code examples, see Using DynamoDB with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Manage DynamoDB Global Secondary Indexes using AWS Command Line Interface v2

The following code example shows how to manage the complete lifecycle of Global Secondary Indexes.

- Create a table with a Global Secondary Index.
- Add a new GSI to an existing table.
- Update (increase) GSI warm throughput.
- Query data using GSIs.
- Delete a GSI.

Bash

AWS CLI with Bash script

Create a table with a Global Secondary Index.

```
# Create a table with a GSI
aws dynamodb create-table \
    --table-name MusicCollection \
    --attribute-definitions \
        AttributeName=Artist,AttributeType=S \
        AttributeName=SongTitle,AttributeType=S \
        AttributeName=AlbumTitle,AttributeType=S \
    --key-schema \
        AttributeName=Artist,KeyType=HASH \
        AttributeName=SongTitle,KeyType=RANGE \
    --billing-mode PAY_PER_REQUEST \
    --global-secondary-indexes \
        "IndexName=AlbumIndex,\
        KeySchema=[{AttributeName=AlbumTitle,KeyType=HASH}],\
        Projection={ProjectionType=ALL}"
```

Add a new (on-demand) GSI to an existing table.

```
# Add a new GSI to an existing table
aws dynamodb update-table \
    --table-name MusicCollection \
    --attribute-definitions \
        AttributeName=Genre,AttributeType=S \
    --global-secondary-index-updates \
        "[{\"Create\":{\"IndexName\":\"GenreIndex\",\
        \"KeySchema\":[{\"AttributeName\":\"Genre\",\"KeyType\":\"HASH\"}],\
        \"Projection\":{\"ProjectionType\":\"ALL\"}}}]"
```

Update (increase) GSI warm throughput.

```
# Increase the warm throughput of a GSI (default values are 12k reads, 4k writes)
aws dynamodb update-table \
    --table-name MusicCollection \
    --global-secondary-index-updates \
        "[{\"Update\":{\"IndexName\":\"AlbumIndex\",\
        \"WarmThroughput\":{\"ReadUnitsPerSecond\":15000,\"WriteUnitsPerSecond
\":6000}}}]"
```

Query data using GSIs.

```
# Query the AlbumIndex GSI
aws dynamodb query \
    --table-name MusicCollection \
    --index-name AlbumIndex \
    --key-condition-expression "AlbumTitle = :album" \
    --expression-attribute-values '{":album":{"S":"Let It Be"}}'

# Query the GenreIndex GSI
aws dynamodb query \
    --table-name MusicCollection \
    --index-name GenreIndex \
    --key-condition-expression "Genre = :genre" \
    --expression-attribute-values '{":genre":{"S":"Jazz"}}'
```

Delete a GSI.

```
# Delete a GSI from a table
```

```
aws dynamodb update-table \
    --table-name MusicCollection \
    --global-secondary-index-updates \
        "[{\"Delete\":{\"IndexName\":\"GenreIndex\"}}]"
```

- For API details, see the following topics in *AWS CLI Command Reference*.
  - CreateTable
  - DeleteTable
  - Query
  - UpdateTable

For a complete list of AWS SDK developer guides and code examples, see Using DynamoDB with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Manage DynamoDB resource-based policies using AWS Command Line Interface v2

The following code example shows how to manage the complete lifecycle of resource-based policies for DynamoDB tables.

- Create a table with a resource policy.
- Get a resource policy.
- Update a resource policy.
- Delete a resource policy.

Bash

AWS CLI with Bash script

Create a table with a resource policy.

```
# Step 1: Create a DynamoDB table
aws dynamodb create-table \
    --table-name MusicCollection \
    --attribute-definitions \
        AttributeName=Artist,AttributeType=S \
```

```
        AttributeName=SongTitle,AttributeType=S \
    --key-schema \
        AttributeName=Artist,KeyType=HASH \
        AttributeName=SongTitle,KeyType=RANGE \
    --billing-mode PAY_PER_REQUEST

# Step 2: Create a resource-based policy document
cat > policy.json << 'EOF'
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "AWS": "arn:aws:iam::123456789012:role/DynamoDBReadOnly"
      },
      "Action": [
        "dynamodb:GetItem",
        "dynamodb:BatchGetItem",
        "dynamodb:Query",
        "dynamodb:Scan"
      ],
      "Resource": "arn:aws:dynamodb:us-west-2:123456789012:table/MusicCollection"
    }
  ]
}
EOF

# Step 3: Attach the resource-based policy to the table
aws dynamodb put-resource-policy \
    --resource-arn arn:aws:dynamodb:us-west-2:123456789012:table/MusicCollection
 \
    --policy file://policy.json
```

Get a resource policy.

```
# Get the resource-based policy attached to a table
aws dynamodb get-resource-policy \
    --resource-arn arn:aws:dynamodb:us-west-2:123456789012:table/MusicCollection
```

Update a resource policy.

```
# Step 1: Create an updated policy document
cat > updated-policy.json << 'EOF'
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "AWS": [
          "arn:aws:iam::123456789012:role/DynamoDBReadOnly",
          "arn:aws:iam::123456789012:role/DynamoDBAnalytics"
        ]
      },
      "Action": [
        "dynamodb:GetItem",
        "dynamodb:BatchGetItem",
        "dynamodb:Query",
        "dynamodb:Scan"
      ],
      "Resource": "arn:aws:dynamodb:us-west-2:123456789012:table/MusicCollection"
    }
  ]
}
EOF

# Step 2: Update the resource-based policy on the table
aws dynamodb put-resource-policy \
    --resource-arn arn:aws:dynamodb:us-west-2:123456789012:table/MusicCollection
 \
    --policy file://updated-policy.json
```

Delete a resource policy.

```
# Delete the resource-based policy from a table
aws dynamodb delete-resource-policy \
    --resource-arn arn:aws:dynamodb:us-west-2:123456789012:table/MusicCollection
```

- For API details, see the following topics in *AWS CLI Command Reference*.
  - CreateTable
  - DeleteResourcePolicy

- GetResourcePolicy
- PutResourcePolicy

For a complete list of AWS SDK developer guides and code examples, see Using DynamoDB with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Monitor performance of Amazon DynamoDB using an AWS SDK

The following code example shows how to configure an application's use of DynamoDB to monitor performance.

Java

SDK for Java 2.x

This example shows how to configure a Java application to monitor the performance of DynamoDB. The application sends metric data to CloudWatch where you can monitor the performance.

For complete source code and instructions on how to set up and run, see the full example on GitHub.

Services used in this example

- CloudWatch
- DynamoDB

For a complete list of AWS SDK developer guides and code examples, see Using DynamoDB with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Perform advanced DynamoDB query operations using an AWS SDK

The following code examples show how to perform advanced query operations in DynamoDB.

- Query tables using various filtering and condition techniques.
- Implement pagination for large result sets.
- Use Global Secondary Indexes for alternate access patterns.

- Apply consistency controls based on application requirements.

Java

SDK for Java 2.x

Query with strongly consistent reads using AWS SDK for Java 2.x.

```
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.QueryRequest;
import software.amazon.awssdk.services.dynamodb.model.QueryResponse;
import software.amazon.awssdk.services.dynamodb.model.ResourceNotFoundException;

import java.util.HashMap;
import java.util.Map;
import java.util.logging.Level;
import java.util.logging.Logger;

    public QueryResponse queryWithConsistentReads(
        final String tableName,
        final String partitionKeyName,
        final String partitionKeyValue,
        final boolean useConsistentRead) {

        CodeSampleUtils.validateTableParameters(tableName, partitionKeyName,
 partitionKeyValue);

        // Create expression attribute names for the column names
        final Map<String, String> expressionAttributeNames = new HashMap<>();
        expressionAttributeNames.put(EXPRESSION_ATTRIBUTE_NAME_PK,
 partitionKeyName);

        // Create expression attribute values for the column values
        final Map<String, AttributeValue> expressionAttributeValues = new
HashMap<>();
        expressionAttributeValues.put(
            EXPRESSION_ATTRIBUTE_VALUE_PK,
            AttributeValue.builder().s(partitionKeyValue).build());

        // Create the query request
        final QueryRequest queryRequest = QueryRequest.builder()
```

```
            .tableName(tableName)
            .keyConditionExpression(KEY_CONDITION_EXPRESSION)
            .expressionAttributeNames(expressionAttributeNames)
            .expressionAttributeValues(expressionAttributeValues)
            .consistentRead(useConsistentRead)
            .build();

        try {
            final QueryResponse response = dynamoDbClient.query(queryRequest);
            LOGGER.log(Level.INFO, "Query successful. Found {0} items",
 response.count());
            return response;
        } catch (ResourceNotFoundException e) {
            LOGGER.log(Level.SEVERE, "Table not found: {0}", tableName);
            throw e;
        } catch (DynamoDbException e) {
            LOGGER.log(Level.SEVERE, "Error querying with consistent reads", e);
            throw e;
        }
    }
```

Query using a Global Secondary Index with AWS SDK for Java 2.x.

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.QueryRequest;
import software.amazon.awssdk.services.dynamodb.model.QueryResponse;
import software.amazon.awssdk.services.dynamodb.model.ResourceNotFoundException;

import java.util.HashMap;
import java.util.Map;

    public QueryResponse queryTable(
        final String tableName, final String partitionKeyName, final String
 partitionKeyValue) {

        CodeSampleUtils.validateTableParameters(tableName, partitionKeyName,
 partitionKeyValue);

        // Create expression attribute names for the column names
```

```
        final Map<String, String> expressionAttributeNames = new HashMap<>();
        expressionAttributeNames.put(EXPRESSION_ATTRIBUTE_NAME_PK,
partitionKeyName);

        // Create expression attribute values for the column values
        final Map<String, AttributeValue> expressionAttributeValues = new
HashMap<>();
        expressionAttributeValues.put(
            EXPRESSION_ATTRIBUTE_VALUE_PK,
            AttributeValue.builder().s(partitionKeyValue).build());

        // Create the query request
        final QueryRequest queryRequest = QueryRequest.builder()
            .tableName(tableName)
            .keyConditionExpression(KEY_CONDITION_EXPRESSION)
            .expressionAttributeNames(expressionAttributeNames)
            .expressionAttributeValues(expressionAttributeValues)
            .build();

        try {
            final QueryResponse response = dynamoDbClient.query(queryRequest);
            System.out.println("Query on base table successful. Found " +
response.count() + " items");
            return response;
        } catch (ResourceNotFoundException e) {
            System.err.format("Error: The Amazon DynamoDB table \"%s\" can't be
found.\n", tableName);
            throw new DynamoDbQueryException("Table not found: " + tableName, e);
        } catch (DynamoDbException e) {
            System.err.println("Error querying base table: " + e.getMessage());
            throw new DynamoDbQueryException("Failed to execute query on base
table", e);
        }
    }

    /**
     * Queries a DynamoDB Global Secondary Index (GSI) by partition key.
     *
     * @param tableName         The name of the DynamoDB table
     * @param indexName         The name of the GSI
     * @param partitionKeyName  The name of the GSI partition key attribute
     * @param partitionKeyValue The value of the GSI partition key to query
     * @return The query response from DynamoDB
     * @throws ResourceNotFoundException if the table or index doesn't exist
```

```
     * @throws DynamoDbException if the query fails
     */
    public QueryResponse queryGlobalSecondaryIndex(
        final String tableName, final String indexName, final String
partitionKeyName, final String partitionKeyValue) {

        CodeSampleUtils.validateTableParameters(tableName, partitionKeyName,
partitionKeyValue);
        CodeSampleUtils.validateStringParameter("Index name", indexName);

        // Create expression attribute names for the column names
        final Map<String, String> expressionAttributeNames = new HashMap<>();
        expressionAttributeNames.put(EXPRESSION_ATTRIBUTE_NAME_IK,
partitionKeyName);

        // Create expression attribute values for the column values
        final Map<String, AttributeValue> expressionAttributeValues = new
HashMap<>();
        expressionAttributeValues.put(
            EXPRESSION_ATTRIBUTE_VALUE_IK,
            AttributeValue.builder().s(partitionKeyValue).build());

        // Create the query request
        final QueryRequest queryRequest = QueryRequest.builder()
            .tableName(tableName)
            .indexName(indexName)
            .keyConditionExpression(GSI_KEY_CONDITION_EXPRESSION)
            .expressionAttributeNames(expressionAttributeNames)
            .expressionAttributeValues(expressionAttributeValues)
            .build();

        try {
            final QueryResponse response = dynamoDbClient.query(queryRequest);
            System.out.println("Query on GSI successful. Found " +
response.count() + " items");
            return response;
        } catch (ResourceNotFoundException e) {
            System.err.format(
                "Error: The Amazon DynamoDB table \"%s\" or index \"%s\" can't be
found.\n", tableName, indexName);
            throw new DynamoDbQueryException("Table or index not found: " +
tableName + "/" + indexName, e);
        } catch (DynamoDbException e) {
            System.err.println("Error querying GSI: " + e.getMessage());
```

```
            throw new DynamoDbQueryException("Failed to execute query on GSI",
 e);
        }
    }
```

Query with pagination using AWS SDK for Java 2.x.

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.QueryRequest;
import software.amazon.awssdk.services.dynamodb.model.QueryResponse;
import software.amazon.awssdk.services.dynamodb.model.ResourceNotFoundException;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

    public List<Map<String, AttributeValue>> queryWithPagination(
        final String tableName, final String partitionKeyName, final String
 partitionKeyValue, final int pageSize) {

        CodeSampleUtils.validateTableParameters(tableName, partitionKeyName,
 partitionKeyValue);
        CodeSampleUtils.validatePositiveInteger("Page size", pageSize);

        // Create expression attribute names for the column names
        final Map<String, String> expressionAttributeNames = new HashMap<>();
        expressionAttributeNames.put(EXPRESSION_ATTRIBUTE_NAME_PK,
 partitionKeyName);

        // Create expression attribute values for the column values
        final Map<String, AttributeValue> expressionAttributeValues = new
 HashMap<>();
        expressionAttributeValues.put(
            EXPRESSION_ATTRIBUTE_VALUE_PK,
            AttributeValue.builder().s(partitionKeyValue).build());

        // Create the query request
        QueryRequest.Builder queryRequestBuilder = QueryRequest.builder()
```

```
            .tableName(tableName)
            .keyConditionExpression(KEY_CONDITION_EXPRESSION)
            .expressionAttributeNames(expressionAttributeNames)
            .expressionAttributeValues(expressionAttributeValues)
            .limit(pageSize);

        // List to store all items from all pages
        final List<Map<String, AttributeValue>> allItems = new ArrayList<>();

        // Map to store the last evaluated key for pagination
        Map<String, AttributeValue> lastEvaluatedKey = null;
        int pageNumber = 1;

        try {
            do {
                // If we have a last evaluated key, use it for the next page
                if (lastEvaluatedKey != null) {
                    queryRequestBuilder.exclusiveStartKey(lastEvaluatedKey);
                }

                // Execute the query
                final QueryResponse response =
dynamoDbClient.query(queryRequestBuilder.build());

                // Process the current page of results
                final List<Map<String, AttributeValue>> pageItems =
response.items();
                allItems.addAll(pageItems);

                // Get the last evaluated key for the next page
                lastEvaluatedKey = response.lastEvaluatedKey();
                if (lastEvaluatedKey != null && lastEvaluatedKey.isEmpty()) {
                    lastEvaluatedKey = null;
                }

                System.out.println("Page " + pageNumber + ": Retrieved " +
pageItems.size() + " items (Running total: "
                    + allItems.size() + ")");

                pageNumber++;

            } while (lastEvaluatedKey != null);
```

```
            System.out.println("Query with pagination complete. Retrieved a total
 of " + allItems.size()
                + " items across " + (pageNumber - 1) + " pages");

            return allItems;
        } catch (ResourceNotFoundException e) {
            System.err.format("Error: The Amazon DynamoDB table \"%s\" can't be
 found.\n", tableName);
            throw e;
        } catch (DynamoDbException e) {
            System.err.println("Error querying with pagination: " +
 e.getMessage());
            throw e;
        }
    }
```

Query with complex filters using AWS SDK for Java 2.x.

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.QueryRequest;
import software.amazon.awssdk.services.dynamodb.model.QueryResponse;
import software.amazon.awssdk.services.dynamodb.model.ResourceNotFoundException;

import java.util.HashMap;
import java.util.Map;
import java.util.logging.Level;
import java.util.logging.Logger;

    public QueryResponse queryWithComplexFilter(
        final String tableName,
        final String partitionKeyName,
        final String partitionKeyValue,
        final String statusAttrName,
        final String activeStatus,
        final String pendingStatus,
        final String priceAttrName,
        final double minPrice,
        final double maxPrice,
        final String categoryAttrName) {
```

```
        // Validate parameters
        CodeSampleUtils.validateTableParameters(tableName, partitionKeyName,
partitionKeyValue);
        CodeSampleUtils.validateStringParameter("Status attribute name",
statusAttrName);
        CodeSampleUtils.validateStringParameter("Active status", activeStatus);
        CodeSampleUtils.validateStringParameter("Pending status", pendingStatus);
        CodeSampleUtils.validateStringParameter("Price attribute name",
priceAttrName);
        CodeSampleUtils.validateStringParameter("Category attribute name",
categoryAttrName);
        CodeSampleUtils.validateNumericRange("Minimum price", minPrice, 0.0,
Double.MAX_VALUE);
        CodeSampleUtils.validateNumericRange("Maximum price", maxPrice, minPrice,
Double.MAX_VALUE);

        // Create expression attribute names for the column names
        final Map<String, String> expressionAttributeNames = new HashMap<>();
        expressionAttributeNames.put("#pk", partitionKeyName);
        expressionAttributeNames.put(EXPRESSION_ATTRIBUTE_NAME_STATUS,
statusAttrName);
        expressionAttributeNames.put(EXPRESSION_ATTRIBUTE_NAME_PRICE,
priceAttrName);
        expressionAttributeNames.put(EXPRESSION_ATTRIBUTE_NAME_CATEGORY,
categoryAttrName);

        // Create expression attribute values for the column values
        final Map<String, AttributeValue> expressionAttributeValues = new
HashMap<>();
        expressionAttributeValues.put(
            ":pkValue", AttributeValue.builder().s(partitionKeyValue).build());
        expressionAttributeValues.put(
            EXPRESSION_ATTRIBUTE_VALUE_ACTIVE,
            AttributeValue.builder().s(activeStatus).build());
        expressionAttributeValues.put(
            EXPRESSION_ATTRIBUTE_VALUE_PENDING,
            AttributeValue.builder().s(pendingStatus).build());
        expressionAttributeValues.put(
            EXPRESSION_ATTRIBUTE_VALUE_MIN_PRICE,
            AttributeValue.builder().n(String.valueOf(minPrice)).build());
        expressionAttributeValues.put(
            EXPRESSION_ATTRIBUTE_VALUE_MAX_PRICE,
            AttributeValue.builder().n(String.valueOf(maxPrice)).build());
```

```
        // Create the query request
        final QueryRequest queryRequest = QueryRequest.builder()
            .tableName(tableName)
            .keyConditionExpression(KEY_CONDITION_EXPRESSION)
            .filterExpression(FILTER_EXPRESSION)
            .expressionAttributeNames(expressionAttributeNames)
            .expressionAttributeValues(expressionAttributeValues)
            .build();

        return dynamoDbClient.query(queryRequest);
    }
```

Query with a dynamically constructed filter expression using AWS SDK for Java 2.x.

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.QueryRequest;
import software.amazon.awssdk.services.dynamodb.model.QueryResponse;
import software.amazon.awssdk.services.dynamodb.model.ResourceNotFoundException;

import java.util.HashMap;
import java.util.Map;

    public static QueryResponse queryWithDynamicFilter(
        final String tableName,
        final String partitionKeyName,
        final String partitionKeyValue,
        final Map<String, Object> filterCriteria,
        final Region region,
        final DynamoDbClient dynamoDbClient) {

        validateParameters(tableName, partitionKeyName, partitionKeyValue,
 filterCriteria);

        DynamoDbClient ddbClient = dynamoDbClient;
        boolean shouldClose = false;

        try {
            if (ddbClient == null) {
```

```
                ddbClient = createClient(region);
                shouldClose = true;
            }

            final QueryWithDynamicFilter queryHelper = new
QueryWithDynamicFilter(ddbClient);
            return queryHelper.queryWithDynamicFilter(tableName,
partitionKeyName, partitionKeyValue, filterCriteria);
        } catch (ResourceNotFoundException e) {
            System.err.println("Table not found: " + tableName);
            throw e;
        } catch (DynamoDbException e) {
            System.err.println("Failed to execute dynamic filter query: " +
e.getMessage());
            throw e;
        } catch (Exception e) {
            System.err.println("Unexpected error during query: " +
e.getMessage());
            throw e;
        } finally {
            if (shouldClose && ddbClient != null) {
                ddbClient.close();
            }
        }
    }

    public static void main(String[] args) {
        final String usage =
            """
                Usage:
                    <tableName> <partitionKeyName> <partitionKeyValue>
<filterAttrName> <filterAttrValue> [region]
                Where:
                    tableName - The Amazon DynamoDB table to query.
                    partitionKeyName - The name of the partition key attribute.
                    partitionKeyValue - The value of the partition key to query.
                    filterAttrName - The name of the attribute to filter on.
                    filterAttrValue - The value to filter by.
                    region (optional) - The AWS region where the table exists.
(Default: us-east-1)
                """;

        if (args.length < 5) {
            System.out.println(usage);
```

```
            System.exit(1);
        }

        final String tableName = args[0];
        final String partitionKeyName = args[1];
        final String partitionKeyValue = args[2];
        final String filterAttrName = args[3];
        final String filterAttrValue = args[4];
        final Region region = args.length > 5 ? Region.of(args[5]) :
Region.US_EAST_1;

        System.out.println("Querying items with dynamic filter: " +
filterAttrName + " = " + filterAttrValue);

        try {
            // Using the builder pattern to create and execute the query
            final QueryResponse response = new DynamicFilterQueryBuilder()
                .withTableName(tableName)
                .withPartitionKeyName(partitionKeyName)
                .withPartitionKeyValue(partitionKeyValue)
                .withFilterCriterion(filterAttrName, filterAttrValue)
                .withRegion(region)
                .execute();

            // Process the results
            System.out.println("Found " + response.count() + " items:");
            response.items().forEach(item -> System.out.println(item));

            // Demonstrate multiple filter criteria
            System.out.println("\nNow querying with multiple filter criteria:");

            Map<String, Object> multipleFilters = new HashMap<>();
            multipleFilters.put(filterAttrName, filterAttrValue);
            multipleFilters.put("status", "active");

            final QueryResponse multiFilterResponse = new
DynamicFilterQueryBuilder()
                .withTableName(tableName)
                .withPartitionKeyName(partitionKeyName)
                .withPartitionKeyValue(partitionKeyValue)
                .withFilterCriteria(multipleFilters)
                .withRegion(region)
                .execute();
```

```
            System.out.println("Found " + multiFilterResponse.count() + " items
 with multiple filters:");
            multiFilterResponse.items().forEach(item ->
 System.out.println(item));

        } catch (IllegalArgumentException e) {
            System.err.println("Invalid input: " + e.getMessage());
            System.exit(1);
        } catch (ResourceNotFoundException e) {
            System.err.println("Table not found: " + tableName);
            System.exit(1);
        } catch (DynamoDbException e) {
            System.err.println("DynamoDB error: " + e.getMessage());
            System.exit(1);
        } catch (Exception e) {
            System.err.println("Unexpected error: " + e.getMessage());
            System.exit(1);
        }
    }
```

Query with a filter expression and limit using AWS SDK for Java 2.x.

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.QueryRequest;
import software.amazon.awssdk.services.dynamodb.model.QueryResponse;
import software.amazon.awssdk.services.dynamodb.model.ResourceNotFoundException;

import java.util.HashMap;
import java.util.Map;
import java.util.logging.Level;
import java.util.logging.Logger;

    public QueryResponse queryWithFilterAndLimit(
        final String tableName,
        final String partitionKeyName,
        final String partitionKeyValue,
        final String filterAttrName,
        final String filterAttrValue,
        final int limit) {
```

```
        CodeSampleUtils.validateTableParameters(tableName, partitionKeyName,
partitionKeyValue);
        CodeSampleUtils.validateStringParameter("Filter attribute name",
filterAttrName);
        CodeSampleUtils.validateStringParameter("Filter attribute value",
filterAttrValue);
        CodeSampleUtils.validatePositiveInteger("Limit", limit);

        // Create expression attribute names for the column names
        final Map<String, String> expressionAttributeNames = new HashMap<>();
        expressionAttributeNames.put(EXPRESSION_ATTRIBUTE_NAME_PK,
partitionKeyName);
        expressionAttributeNames.put(EXPRESSION_ATTRIBUTE_NAME_FILTER,
filterAttrName);

        // Create expression attribute values for the column values
        final Map<String, AttributeValue> expressionAttributeValues = new
HashMap<>();
        expressionAttributeValues.put(
            EXPRESSION_ATTRIBUTE_VALUE_PK,
            AttributeValue.builder().s(partitionKeyValue).build());
        expressionAttributeValues.put(
            EXPRESSION_ATTRIBUTE_VALUE_FILTER,
            AttributeValue.builder().s(filterAttrValue).build());

        // Create the filter expression
        final String filterExpression = "#filterAttr = :filterValue";

        // Create the query request
        final QueryRequest queryRequest = QueryRequest.builder()
            .tableName(tableName)
            .keyConditionExpression(KEY_CONDITION_EXPRESSION)
            .filterExpression(filterExpression)
            .expressionAttributeNames(expressionAttributeNames)
            .expressionAttributeValues(expressionAttributeValues)
            .limit(limit)
            .build();

        try {
            final QueryResponse response = dynamoDbClient.query(queryRequest);
            LOGGER.log(Level.INFO, "Query with filter and limit successful. Found
{0} items", response.count());
            LOGGER.log(
```

```
                Level.INFO, "ScannedCount: {0} (total items evaluated before
 filtering)", response.scannedCount());
            return response;
        } catch (ResourceNotFoundException e) {
            LOGGER.log(Level.SEVERE, "Table not found: {0}", tableName);
            throw e;
        } catch (DynamoDbException e) {
            LOGGER.log(Level.SEVERE, "Error querying with filter and limit: {0}",
 e.getMessage());
            throw e;
        }
    }
```

- For API details, see Query in *AWS SDK for Java 2.x API Reference*.

JavaScript

SDK for JavaScript (v3)

Query with strongly consistent reads using AWS SDK for JavaScript.

```
const { DynamoDBClient, QueryCommand } = require("@aws-sdk/client-dynamodb");

/**
 * Queries a DynamoDB table with configurable read consistency
 *
 * @param {Object} config - AWS SDK configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {string} partitionKeyName - The name of the partition key
 * @param {string} partitionKeyValue - The value of the partition key
 * @param {boolean} useConsistentRead - Whether to use strongly consistent reads
 * @returns {Promise<Object>} - The query response
 */
async function queryWithConsistentRead(
  config,
  tableName,
  partitionKeyName,
  partitionKeyValue,
  useConsistentRead = false
) {
  try {
    // Create DynamoDB client
```

```
    const client = new DynamoDBClient(config);

    // Construct the query input
    const input = {
      TableName: tableName,
      KeyConditionExpression: "#pk = :pkValue",
      ExpressionAttributeNames: {
        "#pk": partitionKeyName
      },
      ExpressionAttributeValues: {
        ":pkValue": { S: partitionKeyValue }
      },
      ConsistentRead: useConsistentRead
    };

    // Execute the query
    const command = new QueryCommand(input);
    return await client.send(command);
  } catch (error) {
    console.error(`Error querying with consistent read: ${error}`);
    throw error;
  }
}
```

Query using a Global Secondary Index with AWS SDK for JavaScript.

```
const { DynamoDBClient, QueryCommand } = require("@aws-sdk/client-dynamodb");

/**
 * Queries a DynamoDB table using the primary key
 *
 * @param {Object} config - AWS SDK configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {string} userId - The user ID to query by (partition key)
 * @returns {Promise<Object>} - The query response
 */
async function queryTable(
  config,
  tableName,
  userId
) {
  try {
```

```
    // Create DynamoDB client
    const client = new DynamoDBClient(config);

    // Construct the query input for the base table
    const input = {
      TableName: tableName,
      KeyConditionExpression: "user_id = :userId",
      ExpressionAttributeValues: {
        ":userId": { S: userId }
      }
    };

    // Execute the query
    const command = new QueryCommand(input);
    return await client.send(command);
  } catch (error) {
    console.error(`Error querying table: ${error}`);
    throw error;
  }
}

/**
 * Queries a DynamoDB Global Secondary Index (GSI)
 *
 * @param {Object} config - AWS SDK configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {string} indexName - The name of the GSI to query
 * @param {string} gameId - The game ID to query by (GSI partition key)
 * @returns {Promise<Object>} - The query response
 */
async function queryGSI(
  config,
  tableName,
  indexName,
  gameId
) {
  try {
    // Create DynamoDB client
    const client = new DynamoDBClient(config);

    // Construct the query input for the GSI
    const input = {
      TableName: tableName,
      IndexName: indexName,
```

```
      KeyConditionExpression: "game_id = :gameId",
      ExpressionAttributeValues: {
        ":gameId": { S: gameId }
      }
    };

    // Execute the query
    const command = new QueryCommand(input);
    return await client.send(command);
  } catch (error) {
    console.error(`Error querying GSI: ${error}`);
    throw error;
  }
}
```

Query with pagination using AWS SDK for JavaScript.

```
/**
 * Example demonstrating how to handle large query result sets in DynamoDB using
 pagination
 *
 * This example shows:
 * - How to use pagination to handle large result sets
 * - How to use LastEvaluatedKey to retrieve the next page of results
 * - How to construct subsequent query requests using ExclusiveStartKey
 */
const { DynamoDBClient, QueryCommand } = require("@aws-sdk/client-dynamodb");

/**
 * Queries a DynamoDB table with pagination to handle large result sets
 *
 * @param {Object} config - AWS SDK configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {string} partitionKeyName - The name of the partition key
 * @param {string} partitionKeyValue - The value of the partition key
 * @param {number} pageSize - Number of items per page
 * @returns {Promise<Array>} - All items from the query
 */
async function queryWithPagination(
  config,
  tableName,
```

```
  partitionKeyName,
  partitionKeyValue,
  pageSize = 25
) {
  try {
    // Create DynamoDB client
    const client = new DynamoDBClient(config);

    // Initialize variables for pagination
    let lastEvaluatedKey = undefined;
    const allItems = [];
    let pageCount = 0;

    // Loop until all pages are retrieved
    do {
      // Construct the query input
      const input = {
        TableName: tableName,
        KeyConditionExpression: "#pk = :pkValue",
        Limit: pageSize,
        ExpressionAttributeNames: {
          "#pk": partitionKeyName
        },
        ExpressionAttributeValues: {
          ":pkValue": { S: partitionKeyValue }
        }
      };

      // Add ExclusiveStartKey if we have a LastEvaluatedKey from a previous
 query
      if (lastEvaluatedKey) {
        input.ExclusiveStartKey = lastEvaluatedKey;
      }

      // Execute the query
      const command = new QueryCommand(input);
      const response = await client.send(command);

      // Process the current page of results
      pageCount++;
      console.log(`Processing page ${pageCount} with ${response.Items.length}
 items`);

      // Add the items from this page to our collection
```

```
      if (response.Items && response.Items.length > 0) {
        allItems.push(...response.Items);
      }

      // Get the LastEvaluatedKey for the next page
      lastEvaluatedKey = response.LastEvaluatedKey;

    } while (lastEvaluatedKey); // Continue until there are no more pages

    console.log(`Query complete. Retrieved ${allItems.length} items in
 ${pageCount} pages.`);
    return allItems;
  } catch (error) {
    console.error(`Error querying with pagination: ${error}`);
    throw error;
  }
}

/**
 * Example usage:
 *
 * // Query all items in the "AWS DynamoDB" forum with pagination
 * const allItems = await queryWithPagination(
 *   { region: "us-west-2" },
 *   "ForumThreads",
 *   "ForumName",
 *   "AWS DynamoDB",
 *   25 // 25 items per page
 * );
 *
 * console.log(`Total items retrieved: ${allItems.length}`);
 *
 * // Notes on pagination:
 * // - LastEvaluatedKey contains the primary key of the last evaluated item
 * // - When LastEvaluatedKey is undefined/null, there are no more items to
 retrieve
 * // - ExclusiveStartKey tells DynamoDB where to start the next page
 * // - Pagination helps manage memory usage for large result sets
 * // - Each page requires a separate network request to DynamoDB
 */

module.exports = { queryWithPagination };
```

Query with complex filters using AWS SDK for JavaScript.

```
const { DynamoDBClient, QueryCommand } = require("@aws-sdk/client-dynamodb");

/**
 * Queries a DynamoDB table with a complex filter expression
 *
 * @param {Object} config - AWS SDK configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {string} partitionKeyName - The name of the partition key
 * @param {string} partitionKeyValue - The value of the partition key
 * @param {number|string} minViews - Minimum number of views for filtering
 * @param {number|string} minReplies - Minimum number of replies for filtering
 * @param {string} requiredTag - Tag that must be present in the item's tags set
 * @returns {Promise<Object>} - The query response
 */
async function queryWithComplexFilter(
  config,
  tableName,
  partitionKeyName,
  partitionKeyValue,
  minViews,
  minReplies,
  requiredTag
) {
  try {
    // Create DynamoDB client
    const client = new DynamoDBClient(config);

    // Construct the query input
    const input = {
      TableName: tableName,
      KeyConditionExpression: "#pk = :pkValue",
      FilterExpression: "views >= :minViews AND replies >= :minReplies AND
 contains(tags, :tag)",
      ExpressionAttributeNames: {
        "#pk": partitionKeyName
      },
      ExpressionAttributeValues: {
        ":pkValue": { S: partitionKeyValue },
        ":minViews": { N: minViews.toString() },
        ":minReplies": { N: minReplies.toString() },
        ":tag": { S: requiredTag }
      }
```

```
    };

    // Execute the query
    const command = new QueryCommand(input);
    return await client.send(command);
  } catch (error) {
    console.error(`Error querying with complex filter: ${error}`);
    throw error;
  }
}
```

Query with a dynamically constructed filter expression using AWS SDK for JavaScript.

```
const { DynamoDBClient, QueryCommand } = require("@aws-sdk/client-dynamodb");

async function queryWithDynamicFilter(
  config,
  tableName,
  partitionKeyName,
  partitionKeyValue,
  sortKeyName,
  sortKeyValue,
  filterParams = {}
) {
  try {
    // Create DynamoDB client
    const client = new DynamoDBClient(config);

    // Initialize filter expression components
    let filterExpressions = [];
    const expressionAttributeValues = {
      ":pkValue": { S: partitionKeyValue },
      ":skValue": { S: sortKeyValue }
    };
    const expressionAttributeNames = {
      "#pk": partitionKeyName,
      "#sk": sortKeyName
    };

    // Add status filter if provided
    if (filterParams.status) {
      filterExpressions.push("status = :status");
```

```
      expressionAttributeValues[":status"] = { S: filterParams.status };
    }

    // Add minimum views filter if provided
    if (filterParams.minViews !== undefined) {
      filterExpressions.push("views >= :minViews");
      expressionAttributeValues[":minViews"] = { N:
 filterParams.minViews.toString() };
    }

    // Add author filter if provided
    if (filterParams.author) {
      filterExpressions.push("author = :author");
      expressionAttributeValues[":author"] = { S: filterParams.author };
    }

    // Construct the query input
    const input = {
      TableName: tableName,
      KeyConditionExpression: "#pk = :pkValue AND #sk = :skValue"
    };

    // Add filter expression if any filters were provided
    if (filterExpressions.length > 0) {
      input.FilterExpression = filterExpressions.join(" AND ");
    }

    // Add expression attribute names and values
    input.ExpressionAttributeNames = expressionAttributeNames;
    input.ExpressionAttributeValues = expressionAttributeValues;

    // Execute the query
    const command = new QueryCommand(input);
    return await client.send(command);
  } catch (error) {
    console.error(`Error querying with dynamic filter: ${error}`);
    throw error;
  }
}
```

- For API details, see Query in *AWS SDK for JavaScript API Reference*.

Python

SDK for Python (Boto3)

Query with strongly consistent reads using AWS SDK for Python (Boto3).

```
import time

import boto3
from boto3.dynamodb.conditions import Key


def query_with_consistent_read(
    table_name,
    partition_key_name,
    partition_key_value,
    sort_key_name=None,
    sort_key_value=None,
    consistent_read=True,
):
    """
    Query a DynamoDB table with the option for strongly consistent reads.

    Args:
        table_name (str): The name of the DynamoDB table.
        partition_key_name (str): The name of the partition key attribute.
        partition_key_value (str): The value of the partition key to query.
        sort_key_name (str, optional): The name of the sort key attribute.
        sort_key_value (str, optional): The value of the sort key to query.
        consistent_read (bool, optional): Whether to use strongly consistent
 reads. Defaults to True.

    Returns:
        dict: The response from DynamoDB containing the query results.
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # Build the key condition expression
    key_condition = Key(partition_key_name).eq(partition_key_value)

    if sort_key_name and sort_key_value:
        key_condition = key_condition & Key(sort_key_name).eq(sort_key_value)
```

```
    # Perform the query with the consistent read option
    response = table.query(KeyConditionExpression=key_condition,
 ConsistentRead=consistent_read)

    return response
```

Query using a Global Secondary Index with AWS SDK for Python (Boto3).

```
import boto3
from boto3.dynamodb.conditions import Key


def query_table(table_name, partition_key_name, partition_key_value):
    """
    Query a DynamoDB table using its primary key.

    Args:
        table_name (str): The name of the DynamoDB table.
        partition_key_name (str): The name of the partition key attribute.
        partition_key_value (str): The value of the partition key to query.

    Returns:
        dict: The response from DynamoDB containing the query results.
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # Perform the query on the table's primary key
    response =
 table.query(KeyConditionExpression=Key(partition_key_name).eq(partition_key_value))

    return response


def query_gsi(table_name, index_name, partition_key_name, partition_key_value):
    """
    Query a Global Secondary Index (GSI) on a DynamoDB table.
```

```
    Args:
        table_name (str): The name of the DynamoDB table.
        index_name (str): The name of the Global Secondary Index.
        partition_key_name (str): The name of the GSI's partition key attribute.
        partition_key_value (str): The value of the GSI's partition key to query.

    Returns:
        dict: The response from DynamoDB containing the query results.
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # Perform the query on the GSI
    response = table.query(
        IndexName=index_name,
 KeyConditionExpression=Key(partition_key_name).eq(partition_key_value)
    )

    return response
```

Query with pagination using AWS SDK for Python (Boto3).

```
import boto3
from boto3.dynamodb.conditions import Key


def query_with_pagination(
    table_name, partition_key_name, partition_key_value, page_size=25,
 max_pages=None
):
    """
    Query a DynamoDB table with pagination to handle large result sets.

    Args:
        table_name (str): The name of the DynamoDB table.
        partition_key_name (str): The name of the partition key attribute.
        partition_key_value (str): The value of the partition key to query.
        page_size (int, optional): The number of items to return per page.
 Defaults to 25.
```

```
        max_pages (int, optional): The maximum number of pages to retrieve. If
None, retrieves all pages.

    Returns:
        list: All items retrieved from the query across all pages.
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # Initialize variables for pagination
    last_evaluated_key = None
    page_count = 0
    all_items = []

    # Paginate through the results
    while True:
        # Check if we've reached the maximum number of pages
        if max_pages is not None and page_count >= max_pages:
            break

        # Prepare the query parameters
        query_params = {
            "KeyConditionExpression":
Key(partition_key_name).eq(partition_key_value),
            "Limit": page_size,
        }

        # Add the ExclusiveStartKey if we have a LastEvaluatedKey from a previous
query
        if last_evaluated_key:
            query_params["ExclusiveStartKey"] = last_evaluated_key

        # Execute the query
        response = table.query(**query_params)

        # Process the current page of results
        items = response.get("Items", [])
        all_items.extend(items)

        # Update pagination tracking
        page_count += 1

        # Get the LastEvaluatedKey for the next page, if any
```

```
        last_evaluated_key = response.get("LastEvaluatedKey")

        # If there's no LastEvaluatedKey, we've reached the end of the results
        if not last_evaluated_key:
            break

    return all_items


def query_with_pagination_generator(
    table_name, partition_key_name, partition_key_value, page_size=25
):
    """
    Query a DynamoDB table with pagination using a generator to handle large
 result sets.
    This approach is memory-efficient as it yields one page at a time.

    Args:
        table_name (str): The name of the DynamoDB table.
        partition_key_name (str): The name of the partition key attribute.
        partition_key_value (str): The value of the partition key to query.
        page_size (int, optional): The number of items to return per page.
 Defaults to 25.

    Yields:
        tuple: A tuple containing (items, page_number, last_page) where:
            - items is a list of items for the current page
            - page_number is the current page number (starting from 1)
            - last_page is a boolean indicating if this is the last page
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # Initialize variables for pagination
    last_evaluated_key = None
    page_number = 0

    # Paginate through the results
    while True:
        # Prepare the query parameters
        query_params = {
            "KeyConditionExpression":
 Key(partition_key_name).eq(partition_key_value),
```

```
            "Limit": page_size,
        }

        # Add the ExclusiveStartKey if we have a LastEvaluatedKey from a previous
 query
        if last_evaluated_key:
            query_params["ExclusiveStartKey"] = last_evaluated_key

        # Execute the query
        response = table.query(**query_params)

        # Get the current page of results
        items = response.get("Items", [])
        page_number += 1

        # Get the LastEvaluatedKey for the next page, if any
        last_evaluated_key = response.get("LastEvaluatedKey")

        # Determine if this is the last page
        is_last_page = last_evaluated_key is None

        # Yield the current page of results
        yield (items, page_number, is_last_page)

        # If there's no LastEvaluatedKey, we've reached the end of the results
        if is_last_page:
            break
```

Query with complex filters using AWS SDK for Python (Boto3).

```
import boto3
from boto3.dynamodb.conditions import Attr, Key


def query_with_complex_filter(
    table_name,
    partition_key_name,
    partition_key_value,
    min_rating=None,
    status_list=None,
```

```
    max_price=None,
):
    """
    Query a DynamoDB table with a complex filter expression.

    Args:
        table_name (str): The name of the DynamoDB table.
        partition_key_name (str): The name of the partition key attribute.
        partition_key_value (str): The value of the partition key to query.
        min_rating (float, optional): Minimum rating value for filtering.
        status_list (list, optional): List of status values to include.
        max_price (float, optional): Maximum price value for filtering.

    Returns:
        dict: The response from DynamoDB containing the query results.
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # Start with the key condition expression
    key_condition = Key(partition_key_name).eq(partition_key_value)

    # Initialize the filter expression and expression attribute values
    filter_expression = None
    expression_attribute_values = {}

    # Build the filter expression based on provided parameters
    if min_rating is not None:
        filter_expression = Attr("rating").gte(min_rating)
        expression_attribute_values[":min_rating"] = min_rating

    if status_list and len(status_list) > 0:
        status_condition = None
        for i, status in enumerate(status_list):
            status_value_name = f":status{i}"
            expression_attribute_values[status_value_name] = status

            if status_condition is None:
                status_condition = Attr("status").eq(status)
            else:
                status_condition = status_condition | Attr("status").eq(status)

        if filter_expression is None:
```

```
            filter_expression = status_condition
        else:
            filter_expression = filter_expression & status_condition

    if max_price is not None:
        price_condition = Attr("price").lte(max_price)
        expression_attribute_values[":max_price"] = max_price

        if filter_expression is None:
            filter_expression = price_condition
        else:
            filter_expression = filter_expression & price_condition

    # Prepare the query parameters
    query_params = {"KeyConditionExpression": key_condition}

    if filter_expression:
        query_params["FilterExpression"] = filter_expression
        if expression_attribute_values:
            query_params["ExpressionAttributeValues"] =
 expression_attribute_values

    # Execute the query
    response = table.query(**query_params)
    return response


def query_with_complex_filter_and_or(
    table_name,
    partition_key_name,
    partition_key_value,
    category=None,
    min_rating=None,
    max_price=None,
):
    """
    Query a DynamoDB table with a complex filter expression using AND and OR
 operators.

    Args:
        table_name (str): The name of the DynamoDB table.
        partition_key_name (str): The name of the partition key attribute.
        partition_key_value (str): The value of the partition key to query.
        category (str, optional): Category value for filtering.
```

```
        min_rating (float, optional): Minimum rating value for filtering.
        max_price (float, optional): Maximum price value for filtering.

    Returns:
        dict: The response from DynamoDB containing the query results.
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # Start with the key condition expression
    key_condition = Key(partition_key_name).eq(partition_key_value)

    # Build a complex filter expression with AND and OR operators
    filter_expression = None
    expression_attribute_values = {}

    # Build the category condition
    if category:
        filter_expression = Attr("category").eq(category)
        expression_attribute_values[":category"] = category

    # Build the rating and price condition (rating >= min_rating OR price <=
max_price)
    rating_price_condition = None

    if min_rating is not None:
        rating_price_condition = Attr("rating").gte(min_rating)
        expression_attribute_values[":min_rating"] = min_rating

    if max_price is not None:
        price_condition = Attr("price").lte(max_price)
        expression_attribute_values[":max_price"] = max_price

        if rating_price_condition is None:
            rating_price_condition = price_condition
        else:
            rating_price_condition = rating_price_condition | price_condition

    # Combine the conditions
    if rating_price_condition:
        if filter_expression is None:
            filter_expression = rating_price_condition
        else:
```

```
            filter_expression = filter_expression & rating_price_condition

    # Prepare the query parameters
    query_params = {"KeyConditionExpression": key_condition}

    if filter_expression:
        query_params["FilterExpression"] = filter_expression
        if expression_attribute_values:
            query_params["ExpressionAttributeValues"] =
 expression_attribute_values

    # Execute the query
    response = table.query(**query_params)
    return response
```

Query with a dynamically constructed filter expression using AWS SDK for Python (Boto3).

```
import boto3
from boto3.dynamodb.conditions import Attr, Key


def query_with_dynamic_filter(
    table_name, partition_key_name, partition_key_value, filter_conditions=None
):
    """
    Query a DynamoDB table with a dynamically constructed filter expression.

    Args:
        table_name (str): The name of the DynamoDB table.
        partition_key_name (str): The name of the partition key attribute.
        partition_key_value (str): The value of the partition key to query.
        filter_conditions (dict, optional): A dictionary of filter conditions
 where
            keys are attribute names and values are dictionaries with 'operator'
 and 'value'.
            Example: {'rating': {'operator': '>=', 'value': 4}, 'status':
 {'operator': '=', 'value': 'active'}}

    Returns:
        dict: The response from DynamoDB containing the query results.
```

```
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # Start with the key condition expression
    key_condition = Key(partition_key_name).eq(partition_key_value)

    # Initialize variables for the filter expression and attribute values
    filter_expression = None
    expression_attribute_values = {":pk_val": partition_key_value}

    # Dynamically build the filter expression if filter conditions are provided
    if filter_conditions:
        for attr_name, condition in filter_conditions.items():
            operator = condition.get("operator")
            value = condition.get("value")
            attr_value_name = f":{attr_name}"
            expression_attribute_values[attr_value_name] = value

            # Create the appropriate filter expression based on the operator
            current_condition = None
            if operator == "=":
                current_condition = Attr(attr_name).eq(value)
            elif operator == "!=":
                current_condition = Attr(attr_name).ne(value)
            elif operator == ">":
                current_condition = Attr(attr_name).gt(value)
            elif operator == ">=":
                current_condition = Attr(attr_name).gte(value)
            elif operator == "<":
                current_condition = Attr(attr_name).lt(value)
            elif operator == "<=":
                current_condition = Attr(attr_name).lte(value)
            elif operator == "contains":
                current_condition = Attr(attr_name).contains(value)
            elif operator == "begins_with":
                current_condition = Attr(attr_name).begins_with(value)

            # Combine with existing filter expression using AND
            if current_condition:
                if filter_expression is None:
                    filter_expression = current_condition
                else:
```

```
                    filter_expression = filter_expression & current_condition

    # Perform the query with the dynamically built filter expression
    query_params = {"KeyConditionExpression": key_condition}

    if filter_expression:
        query_params["FilterExpression"] = filter_expression

    response = table.query(**query_params)
    return response
```

Query with a filter expression and limit using AWS SDK for Python (Boto3).

```
import boto3
from boto3.dynamodb.conditions import Attr, Key


def query_with_filter_and_limit(
    table_name,
    partition_key_name,
    partition_key_value,
    filter_attribute=None,
    filter_value=None,
    limit=10,
):
    """
    Query a DynamoDB table with a filter expression and limit the number of
 results.

    Args:
        table_name (str): The name of the DynamoDB table.
        partition_key_name (str): The name of the partition key attribute.
        partition_key_value (str): The value of the partition key to query.
        filter_attribute (str, optional): The attribute name to filter on.
        filter_value (any, optional): The value to compare against in the filter.
        limit (int, optional): The maximum number of items to evaluate. Defaults
 to 10.

    Returns:
        dict: The response from DynamoDB containing the query results.
```

```
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # Build the key condition expression
    key_condition = Key(partition_key_name).eq(partition_key_value)

    # Prepare the query parameters
    query_params = {"KeyConditionExpression": key_condition, "Limit": limit}

    # Add the filter expression if filter attributes are provided
    if filter_attribute and filter_value is not None:
        query_params["FilterExpression"] =
Attr(filter_attribute).gt(filter_value)
        query_params["ExpressionAttributeValues"] = {":filter_value":
filter_value}

    # Execute the query
    response = table.query(**query_params)
    return response
```

- For API details, see Query in *AWS SDK for Python (Boto3) API Reference*.

For a complete list of AWS SDK developer guides and code examples, see Using DynamoDB with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Perform list operations in DynamoDB with an AWS SDK

The following code examples show how to perform list operations in DynamoDB.

- Add elements to a list attribute.
- Remove elements from a list attribute.
- Update specific elements in a list by index.
- Use list append and list index functions.

Java

SDK for Java 2.x

Demonstrate list operations using AWS SDK for Java 2.x.

```
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.GetItemRequest;
import software.amazon.awssdk.services.dynamodb.model.GetItemResponse;
import software.amazon.awssdk.services.dynamodb.model.UpdateItemRequest;
import software.amazon.awssdk.services.dynamodb.model.UpdateItemResponse;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

    /**
     * Appends items to a list attribute.
     *
     * <p>This method demonstrates how to use the list_append function to add
     * items to the end of a list attribute.
     *
     * @param dynamoDbClient The DynamoDB client
     * @param tableName The name of the DynamoDB table
     * @param key The key of the item to update
     * @param listAttributeName The name of the list attribute
     * @param itemsToAppend The items to append to the list
     * @return The response from DynamoDB
     * @throws DynamoDbException if an error occurs during the operation
     */
    public static UpdateItemResponse appendToList(
        DynamoDbClient dynamoDbClient,
        String tableName,
        Map<String, AttributeValue> key,
        String listAttributeName,
        List<AttributeValue> itemsToAppend) {

        // Create a list value from the items to append
        AttributeValue listValue =
 AttributeValue.builder().l(itemsToAppend).build();
```

```
        // Define the update parameters
        UpdateItemRequest request = UpdateItemRequest.builder()
            .tableName(tableName)
            .key(key)
            .updateExpression("SET #attrName =
list_append(if_not_exists(#attrName, :emptyList), :newItems)")
            .expressionAttributeNames(Map.of("#attrName", listAttributeName))
            .expressionAttributeValues(Map.of(
                ":newItems",
                listValue,
                ":emptyList",
                AttributeValue.builder().l(new
ArrayList<AttributeValue>()).build()))
            .returnValues("UPDATED_NEW")
            .build();

        // Perform the update operation
        return dynamoDbClient.updateItem(request);
    }

    /**
     * Prepends items to a list attribute.
     *
     * <p>This method demonstrates how to use the list_append function to add
     * items to the beginning of a list attribute.
     *
     * @param dynamoDbClient The DynamoDB client
     * @param tableName The name of the DynamoDB table
     * @param key The key of the item to update
     * @param listAttributeName The name of the list attribute
     * @param itemsToPrepend The items to prepend to the list
     * @return The response from DynamoDB
     * @throws DynamoDbException if an error occurs during the operation
     */
    public static UpdateItemResponse prependToList(
        DynamoDbClient dynamoDbClient,
        String tableName,
        Map<String, AttributeValue> key,
        String listAttributeName,
        List<AttributeValue> itemsToPrepend) {

        // Create a list value from the items to prepend
        AttributeValue listValue =
AttributeValue.builder().l(itemsToPrepend).build();
```

```
        // Define the update parameters
        // Note: To prepend, we put the new items first in the list_append
function
        UpdateItemRequest request = UpdateItemRequest.builder()
            .tableName(tableName)
            .key(key)
            .updateExpression("SET #attrName = list_append(:newItems,
if_not_exists(#attrName, :emptyList))")
            .expressionAttributeNames(Map.of("#attrName", listAttributeName))
            .expressionAttributeValues(Map.of(
                ":newItems",
                listValue,
                ":emptyList",
                AttributeValue.builder().l(new
ArrayList<AttributeValue>()).build()))
            .returnValues("UPDATED_NEW")
            .build();

        // Perform the update operation
        return dynamoDbClient.updateItem(request);
    }

    /**
     * Updates a specific element in a list attribute.
     *
     * <p>This method demonstrates how to update a specific element in a list
     * by its index.
     *
     * @param dynamoDbClient The DynamoDB client
     * @param tableName The name of the DynamoDB table
     * @param key The key of the item to update
     * @param listAttributeName The name of the list attribute
     * @param index The index of the element to update
     * @param newValue The new value for the element
     * @return The response from DynamoDB
     * @throws DynamoDbException if an error occurs during the operation
     */
    public static UpdateItemResponse updateListElement(
        DynamoDbClient dynamoDbClient,
        String tableName,
        Map<String, AttributeValue> key,
        String listAttributeName,
        int index,
```

```
        AttributeValue newValue) {

        // Define the update parameters
        UpdateItemRequest request = UpdateItemRequest.builder()
            .tableName(tableName)
            .key(key)
            .updateExpression("SET #attrName[" + index + "] = :newValue")
            .expressionAttributeNames(Map.of("#attrName", listAttributeName))
            .expressionAttributeValues(Map.of(":newValue", newValue))
            .returnValues("UPDATED_NEW")
            .build();

        // Perform the update operation
        return dynamoDbClient.updateItem(request);
    }

    /**
     * Removes a specific element from a list attribute.
     *
     * <p>This method demonstrates how to remove a specific element from a list
     * by its index.
     *
     * @param dynamoDbClient The DynamoDB client
     * @param tableName The name of the DynamoDB table
     * @param key The key of the item to update
     * @param listAttributeName The name of the list attribute
     * @param index The index of the element to remove
     * @return The response from DynamoDB
     * @throws DynamoDbException if an error occurs during the operation
     */
    public static UpdateItemResponse removeListElement(
        DynamoDbClient dynamoDbClient,
        String tableName,
        Map<String, AttributeValue> key,
        String listAttributeName,
        int index) {

        // Define the update parameters
        UpdateItemRequest request = UpdateItemRequest.builder()
            .tableName(tableName)
            .key(key)
            .updateExpression("REMOVE #attrName[" + index + "]")
            .expressionAttributeNames(Map.of("#attrName", listAttributeName))
            .returnValues("UPDATED_NEW")
```

```
            .build();

        // Perform the update operation
        return dynamoDbClient.updateItem(request);
    }

    /**
     * Gets the current value of a list attribute.
     *
     * <p>Helper method to retrieve the current value of a list attribute.
     *
     * @param dynamoDbClient The DynamoDB client
     * @param tableName The name of the DynamoDB table
     * @param key The key of the item to get
     * @param listAttributeName The name of the list attribute
     * @return The list attribute value or null if not found
     * @throws DynamoDbException if an error occurs during the operation
     */
    public static List<AttributeValue> getListAttribute(
        DynamoDbClient dynamoDbClient, String tableName, Map<String,
AttributeValue> key, String listAttributeName) {

        // Define the get parameters
        GetItemRequest request = GetItemRequest.builder()
            .tableName(tableName)
            .key(key)
            .projectionExpression(listAttributeName)
            .build();

        try {
            // Perform the get operation
            GetItemResponse response = dynamoDbClient.getItem(request);

            // Return the list attribute if it exists, otherwise null
            if (response.item() != null &&
response.item().containsKey(listAttributeName)) {
                return response.item().get(listAttributeName).l();
            }

            return null;
        } catch (DynamoDbException e) {
            throw DynamoDbException.builder()
                .message("Failed to get list attribute: " + e.getMessage())
                .cause(e)
```

```
                .build();
        }
    }
```

Example usage of list operations with AWS SDK for Java 2.x.

```
   public static void exampleUsage(DynamoDbClient dynamoDbClient, String
tableName) {
       // Example key
       Map<String, AttributeValue> key = new HashMap<>();
       key.put("ProductId", AttributeValue.builder().s("P12345").build());

       System.out.println("Demonstrating list operations in DynamoDB");

       try {
           // Example 1: Append items to a list
           System.out.println("\nExample 1: Appending items to a list");
           List<AttributeValue> tagsToAppend = List.of(
               AttributeValue.builder().s("Electronics").build(),
               AttributeValue.builder().s("Gadget").build());

           UpdateItemResponse appendResponse = appendToList(dynamoDbClient,
tableName, key, "Tags", tagsToAppend);

           System.out.println("Updated list attribute: " +
appendResponse.attributes());

           // Example 2: Prepend items to a list
           System.out.println("\nExample 2: Prepending items to a list");
           List<AttributeValue> tagsToPrepend = List.of(
               AttributeValue.builder().s("Featured").build(),
               AttributeValue.builder().s("New").build());

           UpdateItemResponse prependResponse = prependToList(dynamoDbClient,
tableName, key, "Tags", tagsToPrepend);

           System.out.println("Updated list attribute: " +
prependResponse.attributes());

           // Example 3: Update a specific element in a list
           System.out.println("\nExample 3: Updating a specific element in a
list");
```

```
            UpdateItemResponse updateResponse = updateListElement(
                dynamoDbClient,
                tableName,
                key,
                "Tags",
                0,
                AttributeValue.builder().s("BestSeller").build());

            System.out.println("Updated list attribute: " +
updateResponse.attributes());

            // Example 4: Remove a specific element from a list
            System.out.println("\nExample 4: Removing a specific element from a
list");
            UpdateItemResponse removeResponse = removeListElement(dynamoDbClient,
tableName, key, "Tags", 1);

            System.out.println("Updated list attribute: " +
removeResponse.attributes());

            // Example 5: Get the current value of a list attribute
            System.out.println("\nExample 5: Getting the current value of a list
attribute");
            List<AttributeValue> currentList = getListAttribute(dynamoDbClient,
tableName, key, "Tags");

            if (currentList != null) {
                System.out.println("Current list attribute:");
                for (int i = 0; i < currentList.size(); i++) {
                    System.out.println("  [" + i + "]: " +
currentList.get(i).s());
                }
            } else {
                System.out.println("List attribute not found");
            }

            // Explain list operations
            System.out.println("\nKey points about DynamoDB list operations:");
            System.out.println("1. Lists are ordered collections of attributes");
            System.out.println("2. Use list_append to add items to a list");
            System.out.println("3. To append items, use list_append(existingList,
newItems)");
            System.out.println("4. To prepend items, use list_append(newItems,
existingList)");
```

```
            System.out.println("5. Use index notation (list[0]) to access or
 update specific elements");
            System.out.println("6. Use REMOVE to delete elements from a list");
            System.out.println("7. List indices are zero-based");
            System.out.println("8. Use if_not_exists to handle the case where the
 list doesn't exist yet");

        } catch (DynamoDbException e) {
            System.err.println("Error: " + e.getMessage());
            e.printStackTrace();
        }
    }
```

- For API details, see UpdateItem in *AWS SDK for Java 2.x API Reference*.

JavaScript

SDK for JavaScript (v3)

Demonstrate list operations using AWS SDK for JavaScript.

```
const { DynamoDBClient } = require("@aws-sdk/client-dynamodb");
const {
  DynamoDBDocumentClient,
  UpdateCommand,
  GetCommand,
  PutCommand
} = require("@aws-sdk/lib-dynamodb");

/**
 * Append elements to a list attribute.
 *
 * This function demonstrates how to use the list_append function to add elements
 * to the end of a list.
 *
 * @param {Object} config - AWS configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {Object} key - The key of the item to update
 * @param {string} listName - The name of the list attribute
 * @param {Array} values - The values to append to the list
 * @returns {Promise<Object>} - The response from DynamoDB
 */
```

```
async function appendToList(
  config,
  tableName,
  key,
  listName,
  values
) {
  // Initialize the DynamoDB client
  const client = new DynamoDBClient(config);
  const docClient = DynamoDBDocumentClient.from(client);

  // Define the update parameters using list_append
  const params = {
    TableName: tableName,
    Key: key,
    UpdateExpression: `SET ${listName} =
 list_append(if_not_exists(${listName}, :empty_list), :values)`,
    ExpressionAttributeValues: {
      ":empty_list": [],
      ":values": values
    },
    ReturnValues: "UPDATED_NEW"
  };

  // Perform the update operation
  const response = await docClient.send(new UpdateCommand(params));

  return response;
}

/**
 * Prepend elements to a list attribute.
 *
 * This function demonstrates how to use the list_append function to add elements
 * to the beginning of a list.
 *
 * @param {Object} config - AWS configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {Object} key - The key of the item to update
 * @param {string} listName - The name of the list attribute
 * @param {Array} values - The values to prepend to the list
 * @returns {Promise<Object>} - The response from DynamoDB
 */
async function prependToList(
```

```
  config,
  tableName,
  key,
  listName,
  values
) {
  // Initialize the DynamoDB client
  const client = new DynamoDBClient(config);
  const docClient = DynamoDBDocumentClient.from(client);

  // Define the update parameters using list_append
  // Note: To prepend, we put the new values first in the list_append function
  const params = {
    TableName: tableName,
    Key: key,
    UpdateExpression: `SET ${listName} = list_append(:values,
 if_not_exists(${listName}, :empty_list))`,
    ExpressionAttributeValues: {
      ":empty_list": [],
      ":values": values
    },
    ReturnValues: "UPDATED_NEW"
  };

  // Perform the update operation
  const response = await docClient.send(new UpdateCommand(params));

  return response;
}

/**
 * Update a specific element in a list by index.
 *
 * This function demonstrates how to update a specific element in a list
 * using the index notation.
 *
 * @param {Object} config - AWS configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {Object} key - The key of the item to update
 * @param {string} listName - The name of the list attribute
 * @param {number} index - The index of the element to update
 * @param {any} value - The new value for the element
 * @returns {Promise<Object>} - The response from DynamoDB
 */
```

```
async function updateListElement(
  config,
  tableName,
  key,
  listName,
  index,
  value
) {
  // Initialize the DynamoDB client
  const client = new DynamoDBClient(config);
  const docClient = DynamoDBDocumentClient.from(client);

  // Define the update parameters using index notation
  const params = {
    TableName: tableName,
    Key: key,
    UpdateExpression: `SET ${listName}[${index}] = :value`,
    ExpressionAttributeValues: {
      ":value": value
    },
    ReturnValues: "UPDATED_NEW"
  };

  // Perform the update operation
  const response = await docClient.send(new UpdateCommand(params));

  return response;
}

/**
 * Remove an element from a list by index.
 *
 * This function demonstrates how to remove a specific element from a list
 * using the REMOVE action with index notation.
 *
 * @param {Object} config - AWS configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {Object} key - The key of the item to update
 * @param {string} listName - The name of the list attribute
 * @param {number} index - The index of the element to remove
 * @returns {Promise<Object>} - The response from DynamoDB
 */
async function removeListElement(
  config,
```

```
  tableName,
  key,
  listName,
  index
) {
  // Initialize the DynamoDB client
  const client = new DynamoDBClient(config);
  const docClient = DynamoDBDocumentClient.from(client);

  // Define the update parameters using REMOVE with index notation
  const params = {
    TableName: tableName,
    Key: key,
    UpdateExpression: `REMOVE ${listName}[${index}]`,
    ReturnValues: "UPDATED_NEW"
  };

  // Perform the update operation
  const response = await docClient.send(new UpdateCommand(params));

  return response;
}

/**
 * Concatenate two lists.
 *
 * This function demonstrates how to concatenate two lists using the list_append
 function.
 *
 * @param {Object} config - AWS configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {Object} key - The key of the item to update
 * @param {string} listName1 - The name of the first list attribute
 * @param {string} listName2 - The name of the second list attribute
 * @param {string} resultListName - The name of the attribute to store the
 concatenated list
 * @returns {Promise<Object>} - The response from DynamoDB
 */
async function concatenateLists(
  config,
  tableName,
  key,
  listName1,
  listName2,
```

```
  resultListName
) {
  // Initialize the DynamoDB client
  const client = new DynamoDBClient(config);
  const docClient = DynamoDBDocumentClient.from(client);

  // Define the update parameters using list_append
  const params = {
    TableName: tableName,
    Key: key,
    UpdateExpression: `SET ${resultListName} =
 list_append(if_not_exists(${listName1}, :empty_list),
 if_not_exists(${listName2}, :empty_list))`,
    ExpressionAttributeValues: {
      ":empty_list": []
    },
    ReturnValues: "UPDATED_NEW"
  };

  // Perform the update operation
  const response = await docClient.send(new UpdateCommand(params));

  return response;
}

/**
 * Create a nested list structure.
 *
 * This function demonstrates how to create and work with nested lists.
 *
 * @param {Object} config - AWS configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {Object} key - The key of the item to update
 * @param {string} listName - The name of the list attribute
 * @param {Array} nestedLists - An array of arrays to create a nested list
 structure
 * @returns {Promise<Object>} - The response from DynamoDB
 */
async function createNestedList(
  config,
  tableName,
  key,
  listName,
  nestedLists
```

```
) {
  // Initialize the DynamoDB client
  const client = new DynamoDBClient(config);
  const docClient = DynamoDBDocumentClient.from(client);

  // Define the update parameters to create a nested list
  const params = {
    TableName: tableName,
    Key: key,
    UpdateExpression: `SET ${listName} = :nested_lists`,
    ExpressionAttributeValues: {
      ":nested_lists": nestedLists
    },
    ReturnValues: "UPDATED_NEW"
  };

  // Perform the update operation
  const response = await docClient.send(new UpdateCommand(params));

  return response;
}

/**
 * Update an element in a nested list.
 *
 * This function demonstrates how to update an element in a nested list
 * using multiple index notations.
 *
 * @param {Object} config - AWS configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {Object} key - The key of the item to update
 * @param {string} listName - The name of the list attribute
 * @param {number} outerIndex - The index in the outer list
 * @param {number} innerIndex - The index in the inner list
 * @param {any} value - The new value for the element
 * @returns {Promise<Object>} - The response from DynamoDB
 */
async function updateNestedListElement(
  config,
  tableName,
  key,
  listName,
  outerIndex,
  innerIndex,
```

```
  value
) {
  // Initialize the DynamoDB client
  const client = new DynamoDBClient(config);
  const docClient = DynamoDBDocumentClient.from(client);

  // Define the update parameters using multiple index notations
  const params = {
    TableName: tableName,
    Key: key,
    UpdateExpression: `SET ${listName}[${outerIndex}][${innerIndex}] = :value`,
    ExpressionAttributeValues: {
      ":value": value
    },
    ReturnValues: "UPDATED_NEW"
  };

  // Perform the update operation
  const response = await docClient.send(new UpdateCommand(params));

  return response;
}

/**
 * Get the current value of an item.
 *
 * Helper function to retrieve the current value of an item.
 *
 * @param {Object} config - AWS configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {Object} key - The key of the item to get
 * @returns {Promise<Object|null>} - The item or null if not found
 */
async function getItem(
  config,
  tableName,
  key
) {
  // Initialize the DynamoDB client
  const client = new DynamoDBClient(config);
  const docClient = DynamoDBDocumentClient.from(client);

  // Define the get parameters
  const params = {
```

```
    TableName: tableName,
    Key: key
  };

  // Perform the get operation
  const response = await docClient.send(new GetCommand(params));

  // Return the item if it exists, otherwise null
  return response.Item || null;
}
```

Example usage of list operations with AWS SDK for JavaScript.

```
/**
 * Example of how to work with lists in DynamoDB.
 */
async function exampleUsage() {
  // Example parameters
  const config = { region: "us-west-2" };
  const tableName = "UserProfiles";
  const key = { UserId: "U12345" };

  console.log("Demonstrating list operations in DynamoDB");

  try {
    // Example 1: Append elements to a list
    console.log("\nExample 1: Appending elements to a list");
    const response1 = await appendToList(
      config,
      tableName,
      key,
      "RecentSearches",
      ["laptop", "headphones", "monitor"]
    );

    console.log("Appended to list:", response1.Attributes);

    // Example 2: Prepend elements to a list
    console.log("\nExample 2: Prepending elements to a list");
    const response2 = await prependToList(
      config,
      tableName,
```

```
      key,
      "RecentSearches",
      ["keyboard", "mouse"]
    );

    console.log("Prepended to list:", response2.Attributes);

    // Get the current state of the item
    let currentItem = await getItem(config, tableName, key);
    console.log("\nCurrent state of RecentSearches:",
currentItem?.RecentSearches);

    // Example 3: Update a specific element in a list
    console.log("\nExample 3: Updating a specific element in a list");
    const response3 = await updateListElement(
      config,
      tableName,
      key,
      "RecentSearches",
      0, // Update the first element
      "mechanical keyboard" // New value
    );

    console.log("Updated list element:", response3.Attributes);

    // Example 4: Remove an element from a list
    console.log("\nExample 4: Removing an element from a list");
    const response4 = await removeListElement(
      config,
      tableName,
      key,
      "RecentSearches",
      2 // Remove the third element
    );

    console.log("List after removing element:", response4.Attributes);

    // Example 5: Create and concatenate lists
    console.log("\nExample 5: Creating and concatenating lists");

    // First, create two separate lists
    await updateWithMultipleActions(
      config,
      tableName,
```

```
      key,
      "SET WishList = :wishlist, SavedItems = :saveditems",
      null,
      {
        ":wishlist": ["gaming laptop", "wireless earbuds"],
        ":saveditems": ["smartphone", "tablet"]
      }
    );

    // Then, concatenate them
    const response5 = await concatenateLists(
      config,
      tableName,
      key,
      "WishList",
      "SavedItems",
      "AllItems"
    );

    console.log("Concatenated lists:", response5.Attributes);

    // Example 6: Create a nested list structure
    console.log("\nExample 6: Creating a nested list structure");
    const response6 = await createNestedList(
      config,
      tableName,
      key,
      "Categories",
      [
        ["Electronics", "Computers", "Accessories"],
        ["Books", "Magazines", "E-books"],
        ["Clothing", "Shoes", "Watches"]
      ]
    );

    console.log("Created nested list:", response6.Attributes);

    // Example 7: Update an element in a nested list
    console.log("\nExample 7: Updating an element in a nested list");
    const response7 = await updateNestedListElement(
      config,
      tableName,
      key,
      "Categories",
```

```
      0, // First inner list
      1, // Second element in that list
      "Laptops" // New value
    );

    console.log("Updated nested list element:", response7.Attributes);

    // Get the final state of the item
    currentItem = await getItem(config, tableName, key);
    console.log("\nFinal state of the item:", JSON.stringify(currentItem, null,
 2));

    // Explain list operations
    console.log("\nKey points about list operations in DynamoDB:");
    console.log("1. Use list_append to add elements to a list");
    console.log("2. To append elements, use list_append(existingList,
 newElements)");
    console.log("3. To prepend elements, use list_append(newElements,
 existingList)");
    console.log("4. Use if_not_exists to handle cases where the list might not
 exist yet");
    console.log("5. Use index notation (list[0]) to access or update specific
 elements");
    console.log("6. Use REMOVE with index notation to remove elements from a
 list");
    console.log("7. Lists can contain elements of different types");
    console.log("8. Lists can be nested (lists of lists)");
    console.log("9. Use multiple index notations (list[0][1]) to access nested
 list elements");

  } catch (error) {
    console.error("Error:", error);
  }
}

/**
 * Helper function for the examples.
 *
 * @param {Object} config - AWS configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {Object} key - The key of the item to update
 * @param {string} updateExpression - The update expression
 * @param {Object} expressionAttributeNames - Expression attribute name
 placeholders
```

```
 * @param {Object} expressionAttributeValues - Expression attribute value
 placeholders
 * @returns {Promise<Object>} - The response from DynamoDB
 */
async function updateWithMultipleActions(
  config,
  tableName,
  key,
  updateExpression,
  expressionAttributeNames,
  expressionAttributeValues
) {
  // Initialize the DynamoDB client
  const client = new DynamoDBClient(config);
  const docClient = DynamoDBDocumentClient.from(client);

  // Prepare the update parameters
  const updateParams = {
    TableName: tableName,
    Key: key,
    UpdateExpression: updateExpression,
    ReturnValues: "UPDATED_NEW"
  };

  // Add expression attribute names if provided
  if (expressionAttributeNames) {
    updateParams.ExpressionAttributeNames = expressionAttributeNames;
  }

  // Add expression attribute values if provided
  if (expressionAttributeValues) {
    updateParams.ExpressionAttributeValues = expressionAttributeValues;
  }

  // Execute the update
  const response = await docClient.send(new UpdateCommand(updateParams));

  return response;
}
```

- For API details, see UpdateItem in *AWS SDK for JavaScript API Reference*.

Python

SDK for Python (Boto3)

Demonstrate list operations using AWS SDK for Python (Boto3).

```
import boto3
import json
from typing import Any, Dict, List, Optional, Union


def create_list_attribute(
    table_name: str, key: Dict[str, Any], list_name: str, list_values: List[Any]
) -> Dict[str, Any]:
    """
    Create a new list attribute or replace an existing one.

    This function demonstrates how to create a new list attribute or replace
    an existing list with new values.

    Args:
        table_name (str): The name of the DynamoDB table.
        key (Dict[str, Any]): The primary key of the item to update.
        list_name (str): The name of the list attribute.
        list_values (List[Any]): The values to set in the list.

    Returns:
        Dict[str, Any]: The response from DynamoDB containing the updated
 attribute values.
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # Use the SET operation to create or replace the list
    response = table.update_item(
        Key=key,
        UpdateExpression=f"SET {list_name} = :list_values",
        ExpressionAttributeValues={":list_values": list_values},
        ReturnValues="UPDATED_NEW",
    )

    return response
```

```
def append_to_list(
    table_name: str, key: Dict[str, Any], list_name: str, values_to_append:
 List[Any]
) -> Dict[str, Any]:
    """
    Append values to the end of a list attribute.

    This function demonstrates how to use the list_append function to add
 elements
    to the end of a list attribute.

    Args:
        table_name (str): The name of the DynamoDB table.
        key (Dict[str, Any]): The primary key of the item to update.
        list_name (str): The name of the list attribute.
        values_to_append (List[Any]): The values to append to the list.

    Returns:
        Dict[str, Any]: The response from DynamoDB containing the updated
 attribute values.
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # Use list_append to add values to the end of the list
    response = table.update_item(
        Key=key,
        UpdateExpression=f"SET {list_name} = list_append({list_name}, :values)",
        ExpressionAttributeValues={":values": values_to_append},
        ReturnValues="UPDATED_NEW",
    )

    return response


def prepend_to_list(
    table_name: str, key: Dict[str, Any], list_name: str, values_to_prepend:
 List[Any]
) -> Dict[str, Any]:
    """
    Prepend values to the beginning of a list attribute.
```

```
    This function demonstrates how to use the list_append function to add
 elements
    to the beginning of a list attribute.

    Args:
        table_name (str): The name of the DynamoDB table.
        key (Dict[str, Any]): The primary key of the item to update.
        list_name (str): The name of the list attribute.
        values_to_prepend (List[Any]): The values to prepend to the list.

    Returns:
        Dict[str, Any]: The response from DynamoDB containing the updated
 attribute values.
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # Use list_append with reversed order to add values to the beginning of the
 list
    response = table.update_item(
        Key=key,
        UpdateExpression=f"SET {list_name} = list_append(:values, {list_name})",
        ExpressionAttributeValues={":values": values_to_prepend},
        ReturnValues="UPDATED_NEW",
    )

    return response


def update_list_element(
    table_name: str, key: Dict[str, Any], list_name: str, index: int, new_value:
 Any
) -> Dict[str, Any]:
    """
    Update a specific element in a list attribute.

    This function demonstrates how to update a specific element in a list
 attribute
    using the index notation.

    Args:
        table_name (str): The name of the DynamoDB table.
        key (Dict[str, Any]): The primary key of the item to update.
```

```
        list_name (str): The name of the list attribute.
        index (int): The zero-based index of the element to update.
        new_value (Any): The new value for the element.

    Returns:
        Dict[str, Any]: The response from DynamoDB containing the updated
 attribute values.
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # Use the index notation to update a specific element
    response = table.update_item(
        Key=key,
        UpdateExpression=f"SET {list_name}[{index}] = :value",
        ExpressionAttributeValues={":value": new_value},
        ReturnValues="UPDATED_NEW",
    )

    return response


def remove_list_element(
    table_name: str, key: Dict[str, Any], list_name: str, index: int
) -> Dict[str, Any]:
    """
    Remove a specific element from a list attribute.

    This function demonstrates how to remove a specific element from a list
 attribute
    using the REMOVE action with index notation.

    Args:
        table_name (str): The name of the DynamoDB table.
        key (Dict[str, Any]): The primary key of the item to update.
        list_name (str): The name of the list attribute.
        index (int): The zero-based index of the element to remove.

    Returns:
        Dict[str, Any]: The response from DynamoDB containing the updated
 attribute values.
    """
    # Initialize the DynamoDB resource
```

```
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # Use the REMOVE action with index notation to remove a specific element
    response = table.update_item(
        Key=key, UpdateExpression=f"REMOVE {list_name}[{index}]",
 ReturnValues="UPDATED_NEW"
    )

    return response


def update_nested_list_element(
    table_name: str, key: Dict[str, Any], path: str, new_value: Any
) -> Dict[str, Any]:
    """
    Update an element in a nested list structure.

    This function demonstrates how to update an element in a nested list
 structure
    using expression attribute names for the path components.

    Args:
        table_name (str): The name of the DynamoDB table.
        key (Dict[str, Any]): The primary key of the item to update.
        path (str): The path to the nested element (e.g., "parent[0].child[1]").
        new_value (Any): The new value for the element.

    Returns:
        Dict[str, Any]: The response from DynamoDB containing the updated
 attribute values.
    """
    # Define a type for path parts
    path_part = Dict[str, Union[str, int]]
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # Parse the path to extract attribute names and indices
    path_parts: List[path_part] = []
    current_part = ""
    in_bracket = False

    for char in path:
```

```
        if char == "[":
            if current_part:
                path_parts.append({"type": "attribute", "value": current_part})
                current_part = ""
            in_bracket = True
        elif char == "]":
            if current_part:
                # Fix for mypy: Use a properly typed dictionary with Union type
                path_parts.append({"type": "index", "value": int(current_part)})
                current_part = ""
            in_bracket = False
        elif char == "." and not in_bracket:
            if current_part:
                path_parts.append({"type": "attribute", "value": current_part})
                current_part = ""
        else:
            current_part += char

    if current_part:
        path_parts.append({"type": "attribute", "value": current_part})

    # Build the update expression and attribute names
    update_expression = "SET "
    expression_attribute_names = {}

    # Build the path expression
    path_expression = ""
    for i, part in enumerate(path_parts):
        if part["type"] == "attribute":
            name_placeholder = f"#attr{i}"
            expression_attribute_names[name_placeholder] = part["value"]

            if path_expression:
                path_expression += "."
            path_expression += name_placeholder
        elif part["type"] == "index":
            path_expression += f"[{part['value']}]"

    # Complete the update expression
    update_expression += f"{path_expression} = :value"

    # Execute the update
    response = table.update_item(
        Key=key,
```

```
        UpdateExpression=update_expression,
        ExpressionAttributeNames=expression_attribute_names,
        ExpressionAttributeValues={":value": new_value},
        ReturnValues="UPDATED_NEW",
    )

    return response


def create_list_if_not_exists(
    table_name: str, key: Dict[str, Any], list_name: str, default_values:
 List[Any]
) -> Dict[str, Any]:
    """
    Create a list attribute if it doesn't exist.

    This function demonstrates how to use if_not_exists to create a list
 attribute
    with default values if it doesn't already exist.

    Args:
        table_name (str): The name of the DynamoDB table.
        key (Dict[str, Any]): The primary key of the item to update.
        list_name (str): The name of the list attribute.
        default_values (List[Any]): The default values for the list.

    Returns:
        Dict[str, Any]: The response from DynamoDB containing the updated
 attribute values.
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # Use if_not_exists to create the list if it doesn't exist
    response = table.update_item(
        Key=key,
        UpdateExpression=f"SET {list_name} =
 if_not_exists({list_name}, :default)",
        ExpressionAttributeValues={":default": default_values},
        ReturnValues="UPDATED_NEW",
    )

    return response
```

```
def append_to_list_safely(
    table_name: str,
    key: Dict[str, Any],
    list_name: str,
    values_to_append: List[Any],
    default_values: Optional[List[Any]] = None,
) -> Dict[str, Any]:
    """
    Append values to a list, creating it if it doesn't exist.

    This function demonstrates how to safely append values to a list attribute,
    creating the list with default values if it doesn't exist.

    Args:
        table_name (str): The name of the DynamoDB table.
        key (Dict[str, Any]): The primary key of the item to update.
        list_name (str): The name of the list attribute.
        values_to_append (List[Any]): The values to append to the list.
        default_values (Optional[List[Any]]): The default values if the list
 doesn't exist.
            If not provided, values_to_append will be used as the default.

    Returns:
        Dict[str, Any]: The response from DynamoDB containing the updated
 attribute values.
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # If default_values is not provided, use values_to_append
    if default_values is None:
        default_values = values_to_append

    # Use if_not_exists with list_append to safely append to the list
    response = table.update_item(
        Key=key,
        UpdateExpression=f"SET {list_name} =
 list_append(if_not_exists({list_name}, :default), :values)",
        ExpressionAttributeValues={
            ":default": default_values if default_values else [],
            ":values": values_to_append,
```

```
        },
        ReturnValues="UPDATED_NEW",
    )

    return response
```

Example usage of list operations with AWS SDK for Python (Boto3).

```
def example_usage():
    """Example of how to use list operations in DynamoDB."""
    # Example parameters
    table_name = "UserData"
    key = {"UserId": "user123"}

    print("Example 1: Creating a list attribute")
    try:
        response = create_list_attribute(
            table_name=table_name,
            key=key,
            list_name="Interests",
            list_values=["Reading", "Hiking", "Photography"],
        )
        print(
            f"List attribute created successfully:
 {json.dumps(response.get('Attributes', {}), default=str)}"
        )
    except Exception as e:
        print(f"Error creating list attribute: {e}")

    print("\nExample 2: Appending values to a list")
    try:
        response = append_to_list(
            table_name=table_name,
            key=key,
            list_name="Interests",
            values_to_append=["Cooking", "Gardening"],
        )
        print(
            f"Values appended to list successfully:
 {json.dumps(response.get('Attributes', {}), default=str)}"
```

```
        )
    except Exception as e:
        print(f"Error appending to list: {e}")

    print("\nExample 3: Prepending values to a list")
    try:
        response = prepend_to_list(
            table_name=table_name,
            key=key,
            list_name="Interests",
            values_to_prepend=["Travel", "Music"],
        )
        print(
            f"Values prepended to list successfully:
{json.dumps(response.get('Attributes', {}), default=str)}"
        )
    except Exception as e:
        print(f"Error prepending to list: {e}")

    print("\nExample 4: Updating a specific list element")
    try:
        response = update_list_element(
            table_name=table_name,
            key=key,
            list_name="Interests",
            index=2,
            new_value="Mountain Hiking",
        )
        print(
            f"List element updated successfully:
{json.dumps(response.get('Attributes', {}), default=str)}"
        )
    except Exception as e:
        print(f"Error updating list element: {e}")

    print("\nExample 5: Removing a list element")
    try:
        response = remove_list_element(
            table_name=table_name, key=key, list_name="Interests", index=0
        )
        print(
            f"List element removed successfully:
{json.dumps(response.get('Attributes', {}), default=str)}"
        )
```

```
    except Exception as e:
        print(f"Error removing list element: {e}")

    print("\nExample 6: Working with nested lists")
    try:
        # First, create an item with a nested structure
        dynamodb = boto3.resource("dynamodb")
        table = dynamodb.Table(table_name)

        table.update_item(
            Key={"UserId": "user456"},
            UpdateExpression="SET #skills = :skills",
            ExpressionAttributeNames={"#skills": "Skills"},
            ExpressionAttributeValues={
                ":skills": [
                    {"Category": "Programming", "Languages": ["Python", "Java",
"JavaScript"]},
                    {"Category": "Database", "Systems": ["DynamoDB", "MongoDB",
"PostgreSQL"]},
                ]
            },
        )

        # Now update a nested element
        response = update_nested_list_element(
            table_name=table_name,
            key={"UserId": "user456"},
            path="Skills[0].Languages[1]",
            new_value="TypeScript",
        )
        print(
            f"Nested list element updated successfully:
{json.dumps(response.get('Attributes', {}), default=str)}"
        )
    except Exception as e:
        print(f"Error working with nested lists: {e}")

    print("\nExample 7: Creating a list if it doesn't exist")
    try:
        response = create_list_if_not_exists(
            table_name=table_name,
            key={"UserId": "user789"},
            list_name="Preferences",
            default_values=["Default1", "Default2", "Default3"],
```

```
        )
        print(
            f"List created with default values:
{json.dumps(response.get('Attributes', {}), default=str)}"
        )
    except Exception as e:
        print(f"Error creating list with default values: {e}")

    print("\nExample 8: Safely appending to a list")
    try:
        response = append_to_list_safely(
            table_name=table_name,
            key={"UserId": "user789"},
            list_name="Notifications",
            values_to_append=["New message received"],
            default_values=[],
        )
        print(f"Safely appended to list: {json.dumps(response.get('Attributes',
{}), default=str)}")
    except Exception as e:
        print(f"Error safely appending to list: {e}")

    print("\nKey Points About Working with Lists in DynamoDB:")
    print("1. Lists are ordered collections of elements that can be of different
types")
    print("2. Use the SET operation with direct assignment to create or replace a
list")
    print("3. Use list_append() to add elements to a list without replacing the
entire list")
    print("4. To append to the end: list_append(list_name, :values)")
    print("5. To prepend to the beginning: list_append(:values, list_name)")
    print("6. Use index notation list_name[index] to access or update specific
elements")
    print("7. Use the REMOVE action with index notation to remove specific
elements")
    print("8. Lists can contain nested structures like maps and other lists")
    print("9. Use if_not_exists() to create a list with default values if it
doesn't exist")
    print("10. List indices are zero-based (the first element is at index 0)")
    print("11. Attempting to access an index beyond the list bounds will result
in an error")
```

- For API details, see UpdateItem in *AWS SDK for Python (Boto3) API Reference*.

For a complete list of AWS SDK developer guides and code examples, see Using DynamoDB with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Perform map operations in DynamoDB with an AWS SDK

The following code examples show how to perform map operations in DynamoDB.

- Add and update nested attributes in map structures.
- Remove specific fields from maps.
- Work with deeply nested map attributes.

Java

SDK for Java 2.x

Demonstrate map operations using AWS SDK for Java 2.x.

```
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.GetItemRequest;
import software.amazon.awssdk.services.dynamodb.model.GetItemResponse;
import software.amazon.awssdk.services.dynamodb.model.UpdateItemRequest;
import software.amazon.awssdk.services.dynamodb.model.UpdateItemResponse;

import java.util.HashMap;
import java.util.List;
import java.util.Map;

    /**
     * Updates a map attribute that may not exist.
     *
     * <p>This method demonstrates how to safely update a map attribute
     * by using if_not_exists to handle the case where the map doesn't exist yet.
     *
     * @param dynamoDbClient The DynamoDB client
```

```
     * @param tableName The name of the DynamoDB table
     * @param key The key of the item to update
     * @param mapName The name of the map attribute
     * @param mapKey The key within the map to update
     * @param value The value to set
     * @return The response from DynamoDB
     * @throws DynamoDbException if an error occurs during the operation
     */
    public static UpdateItemResponse updateMapAttributeSafe(
        DynamoDbClient dynamoDbClient,
        String tableName,
        Map<String, AttributeValue> key,
        String mapName,
        String mapKey,
        AttributeValue value) {

        // Create an empty map to use if the map doesn't exist
        Map<String, AttributeValue> emptyMap = new HashMap<>();
        AttributeValue emptyMapValue =
AttributeValue.builder().m(emptyMap).build();

        // Define the update parameters
        UpdateItemRequest request = UpdateItemRequest.builder()
            .tableName(tableName)
            .key(key)
            .updateExpression("SET #mapName = if_not_exists(#mapName, :emptyMap),
#mapName.#mapKey = :value")
            .expressionAttributeNames(Map.of(
                "#mapName", mapName,
                "#mapKey", mapKey))
            .expressionAttributeValues(Map.of(
                ":value",
                value,
                ":emptyMap",
                AttributeValue.builder().m(new HashMap<>()).build()))
            .returnValues("UPDATED_NEW")
            .build();

        // Perform the update operation
        return dynamoDbClient.updateItem(request);
    }

    /**
     * Adds an attribute to a nested map.
```

```
     *
     * <p>This method demonstrates how to update a nested attribute without
     * overwriting the entire map.
     *
     * @param dynamoDbClient The DynamoDB client
     * @param tableName The name of the DynamoDB table
     * @param key The key of the item to update
     * @param path The path to the nested attribute as a list
     * @param value The value to set
     * @return The response from DynamoDB
     * @throws DynamoDbException if an error occurs during the operation
     */
    public static UpdateItemResponse addToNestedMap(
        DynamoDbClient dynamoDbClient,
        String tableName,
        Map<String, AttributeValue> key,
        List<String> path,
        AttributeValue value) {

        // Create expression attribute names for each part of the path
        Map<String, String> expressionAttributeNames = new HashMap<>();
        for (int i = 0; i < path.size(); i++) {
            expressionAttributeNames.put("#attr" + i, path.get(i));
        }

        // Build the attribute path using the expression attribute names
        StringBuilder attributePathExpression = new StringBuilder();
        for (int i = 0; i < path.size(); i++) {
            if (i > 0) {
                attributePathExpression.append(".");
            }
            attributePathExpression.append("#attr").append(i);
        }

        // Define the update parameters
        UpdateItemRequest request = UpdateItemRequest.builder()
            .tableName(tableName)
            .key(key)
            .updateExpression("SET " + attributePathExpression.toString() + "
= :value")
            .expressionAttributeNames(expressionAttributeNames)
            .expressionAttributeValues(Map.of(":value", value))
            .returnValues("UPDATED_NEW")
            .build();
```

```
        // Perform the update operation
        return dynamoDbClient.updateItem(request);
    }

    /**
     * Removes an attribute from a map.
     *
     * <p>This method demonstrates how to remove a specific attribute from a map.
     *
     * @param dynamoDbClient The DynamoDB client
     * @param tableName The name of the DynamoDB table
     * @param key The key of the item to update
     * @param mapName The name of the map attribute
     * @param mapKey The key within the map to remove
     * @return The response from DynamoDB
     * @throws DynamoDbException if an error occurs during the operation
     */
    public static UpdateItemResponse removeMapAttribute(
        DynamoDbClient dynamoDbClient,
        String tableName,
        Map<String, AttributeValue> key,
        String mapName,
        String mapKey) {

        // Define the update parameters
        UpdateItemRequest request = UpdateItemRequest.builder()
            .tableName(tableName)
            .key(key)
            .updateExpression("REMOVE #mapName.#mapKey")
            .expressionAttributeNames(Map.of(
                "#mapName", mapName,
                "#mapKey", mapKey))
            .returnValues("UPDATED_NEW")
            .build();

        // Perform the update operation
        return dynamoDbClient.updateItem(request);
    }

    /**
     * Creates a map with multiple attributes in a single operation.
     *
     * <p>This method demonstrates how to create a map with multiple attributes
```

```
     * in a single update operation.
     *
     * @param dynamoDbClient The DynamoDB client
     * @param tableName The name of the DynamoDB table
     * @param key The key of the item to update
     * @param mapName The name of the map attribute
     * @param attributes The attributes to set in the map
     * @return The response from DynamoDB
     * @throws DynamoDbException if an error occurs during the operation
     */
    public static UpdateItemResponse createMapWithAttributes(
        DynamoDbClient dynamoDbClient,
        String tableName,
        Map<String, AttributeValue> key,
        String mapName,
        Map<String, AttributeValue> attributes) {

        // Create a map value from the attributes
        AttributeValue mapValue = AttributeValue.builder().m(attributes).build();

        // Define the update parameters
        UpdateItemRequest request = UpdateItemRequest.builder()
            .tableName(tableName)
            .key(key)
            .updateExpression("SET #mapName = :mapValue")
            .expressionAttributeNames(Map.of("#mapName", mapName))
            .expressionAttributeValues(Map.of(":mapValue", mapValue))
            .returnValues("UPDATED_NEW")
            .build();

        // Perform the update operation
        return dynamoDbClient.updateItem(request);
    }

    /**
     * Gets the current value of a map attribute.
     *
     * <p>Helper method to retrieve the current value of a map attribute.
     *
     * @param dynamoDbClient The DynamoDB client
     * @param tableName The name of the DynamoDB table
     * @param key The key of the item to get
     * @param mapName The name of the map attribute
     * @return The map attribute value or null if not found
```

```
     * @throws DynamoDbException if an error occurs during the operation
     */
    public static Map<String, AttributeValue> getMapAttribute(
        DynamoDbClient dynamoDbClient, String tableName, Map<String,
AttributeValue> key, String mapName) {

        // Define the get parameters
        GetItemRequest request = GetItemRequest.builder()
            .tableName(tableName)
            .key(key)
            .projectionExpression(mapName)
            .build();

        try {
            // Perform the get operation
            GetItemResponse response = dynamoDbClient.getItem(request);

            // Return the map attribute if it exists, otherwise null
            if (response.item() != null && response.item().containsKey(mapName))
{
                return response.item().get(mapName).m();
            }

            return null;
        } catch (DynamoDbException e) {
            throw DynamoDbException.builder()
                .message("Failed to get map attribute: " + e.getMessage())
                .cause(e)
                .build();
        }
    }
```

Example usage of map operations with AWS SDK for Java 2.x.

```
    public static void exampleUsage(DynamoDbClient dynamoDbClient, String
tableName) {
        // Example key
        Map<String, AttributeValue> key = new HashMap<>();
        key.put("ProductId", AttributeValue.builder().s("P12345").build());

        System.out.println("Demonstrating map operations in DynamoDB");
```

```
        try {
            // Example 1: Create a map with multiple attributes
            System.out.println("\nExample 1: Creating a map with multiple
attributes");
            Map<String, AttributeValue> productDetails = new HashMap<>();
            productDetails.put("Color",
AttributeValue.builder().s("Red").build());
            productDetails.put("Weight",
AttributeValue.builder().n("2.5").build());
            productDetails.put(
                "Dimensions", AttributeValue.builder().s("10x20x5").build());

            UpdateItemResponse createResponse =
                createMapWithAttributes(dynamoDbClient, tableName, key,
"Details", productDetails);

            System.out.println("Created map attribute: " +
createResponse.attributes());

            // Example 2: Update a specific attribute in a map
            System.out.println("\nExample 2: Updating a specific attribute in a
map");
            UpdateItemResponse updateResponse = updateMapAttributeSafe(
                dynamoDbClient,
                tableName,
                key,
                "Details",
                "Color",
                AttributeValue.builder().s("Blue").build());

            System.out.println("Updated map attribute: " +
updateResponse.attributes());

            // Example 3: Add an attribute to a nested map
            System.out.println("\nExample 3: Adding an attribute to a nested
map");
            UpdateItemResponse nestedResponse = addToNestedMap(
                dynamoDbClient,
                tableName,
                key,
                List.of("Specifications", "Technical", "Resolution"),
                AttributeValue.builder().s("1920x1080").build());
```

```
            System.out.println("Added to nested map: " +
nestedResponse.attributes());

            // Example 4: Remove an attribute from a map
            System.out.println("\nExample 4: Removing an attribute from a map");
            UpdateItemResponse removeResponse =
                removeMapAttribute(dynamoDbClient, tableName, key, "Details",
"Dimensions");

            System.out.println("Updated map after removal: " +
removeResponse.attributes());

            // Example 5: Get the current value of a map attribute
            System.out.println("\nExample 5: Getting the current value of a map
attribute");
            Map<String, AttributeValue> currentMap =
getMapAttribute(dynamoDbClient, tableName, key, "Details");

            if (currentMap != null) {
                System.out.println("Current map attribute:");
                for (Map.Entry<String, AttributeValue> entry :
currentMap.entrySet()) {
                    System.out.println("  " + entry.getKey() + ": " +
entry.getValue());
                }
            } else {
                System.out.println("Map attribute not found");
            }

            // Explain map operations
            System.out.println("\nKey points about DynamoDB map operations:");
            System.out.println("1. Maps are unordered collections of name-value
pairs");
            System.out.println("2. Use dot notation (map.key) to access or update
specific attributes");
            System.out.println("3. You can update individual attributes without
overwriting the entire map");
            System.out.println("4. Maps can be nested to create complex data
structures");
            System.out.println("5. Use REMOVE to delete attributes from a map");
            System.out.println("6. You can create a map with multiple attributes
in a single operation");
            System.out.println("7. Map keys are case-sensitive");
```

```
        } catch (DynamoDbException e) {
            System.err.println("Error: " + e.getMessage());
            e.printStackTrace();
        }
    }
```

- For API details, see UpdateItem in *AWS SDK for Java 2.x API Reference*.

JavaScript

SDK for JavaScript (v3)

Demonstrate map operations using AWS SDK for JavaScript.

```
/**
 * Example of updating map attributes in DynamoDB.
 *
 * This module demonstrates how to update map attributes that may not exist,
 * how to update nested attributes, and how to handle various map update
 scenarios.
 */

const { DynamoDBClient } = require("@aws-sdk/client-dynamodb");
const {
  DynamoDBDocumentClient,
  UpdateCommand,
  GetCommand
} = require("@aws-sdk/lib-dynamodb");

/**
 * Update a map attribute safely, handling the case where the map might not
 exist.
 *
 * This function demonstrates using the if_not_exists function to safely update
 * a map attribute that might not exist yet.
 *
 * @param {Object} config - AWS configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {Object} key - The key of the item to update
 * @param {string} mapName - The name of the map attribute
 * @param {string} mapKey - The key within the map to update
```

```
 * @param {any} value - The value to set
 * @returns {Promise<Object>} - The response from DynamoDB
 */
async function updateMapAttributeSafe(
  config,
  tableName,
  key,
  mapName,
  mapKey,
  value
) {
  // Initialize the DynamoDB client
  const client = new DynamoDBClient(config);
  const docClient = DynamoDBDocumentClient.from(client);

  // Define the update parameters using SET with if_not_exists
  const params = {
    TableName: tableName,
    Key: key,
    UpdateExpression: `SET ${mapName}.${mapKey} = :value`,
    ExpressionAttributeValues: {
      ":value": value
    },
    ReturnValues: "UPDATED_NEW"
  };

  try {
    // Perform the update operation
    const response = await docClient.send(new UpdateCommand(params));
    return response;
  } catch (error) {
    // If the error is because the map doesn't exist, create it
    if (error.name === "ValidationException" &&
        error.message.includes("The document path provided in the update
 expression is invalid")) {

      // Create the map with the specified key-value pair
      const createParams = {
        TableName: tableName,
        Key: key,
        UpdateExpression: `SET ${mapName} = :map`,
        ExpressionAttributeValues: {
          ":map": { [mapKey]: value }
        },
```

```
        ReturnValues: "UPDATED_NEW"
      };

      return await docClient.send(new UpdateCommand(createParams));
    }

    // Re-throw other errors
    throw error;
  }
}

/**
 * Update a map attribute using the if_not_exists function.
 *
 * This function demonstrates a more elegant approach using if_not_exists
 * to handle the case where the map doesn't exist yet.
 *
 * @param {Object} config - AWS configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {Object} key - The key of the item to update
 * @param {string} mapName - The name of the map attribute
 * @param {string} mapKey - The key within the map to update
 * @param {any} value - The value to set
 * @returns {Promise<Object>} - The response from DynamoDB
 */
async function updateMapAttributeWithIfNotExists(
  config,
  tableName,
  key,
  mapName,
  mapKey,
  value
) {
  // Initialize the DynamoDB client
  const client = new DynamoDBClient(config);
  const docClient = DynamoDBDocumentClient.from(client);

  // Define the update parameters using SET with if_not_exists
  const params = {
    TableName: tableName,
    Key: key,
    UpdateExpression: `SET ${mapName} = if_not_exists(${mapName}, :emptyMap),
 ${mapName}.${mapKey} = :value`,
    ExpressionAttributeValues: {
```

```
      ":emptyMap": {},
      ":value": value
    },
    ReturnValues: "UPDATED_NEW"
  };

  // Perform the update operation
  const response = await docClient.send(new UpdateCommand(params));

  return response;
}

/**
 * Add a value to a deeply nested map, creating parent maps if they don't exist.
 *
 * This function demonstrates how to update a deeply nested attribute,
 * creating any parent maps that don't exist along the way.
 *
 * @param {Object} config - AWS configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {Object} key - The key of the item to update
 * @param {string[]} path - The path to the nested attribute as an array of keys
 * @param {any} value - The value to set
 * @returns {Promise<Object>} - The response from DynamoDB
 */
async function addToNestedMap(
  config,
  tableName,
  key,
  path,
  value
) {
  // Initialize the DynamoDB client
  const client = new DynamoDBClient(config);
  const docClient = DynamoDBDocumentClient.from(client);

  // Build the update expression and expression attribute values
  let updateExpression = "SET";
  const expressionAttributeValues = {};

  // For each level in the path, create a map if it doesn't exist
  for (let i = 0; i < path.length; i++) {
    const currentPath = path.slice(0, i + 1).join(".");
    const parentPath = i > 0 ? path.slice(0, i).join(".") : null;
```

```
    if (parentPath) {
      updateExpression += ` ${parentPath} =
 if_not_exists(${parentPath}, :emptyMap${i}),`;
      expressionAttributeValues[`:emptyMap${i}`] = {};
    }
  }

  // Set the final value
  const fullPath = path.join(".");
  updateExpression += ` ${fullPath} = :value`;
  expressionAttributeValues[":value"] = value;

  // Define the update parameters
  const params = {
    TableName: tableName,
    Key: key,
    UpdateExpression: updateExpression,
    ExpressionAttributeValues: expressionAttributeValues,
    ReturnValues: "UPDATED_NEW"
  };

  // Perform the update operation
  const response = await docClient.send(new UpdateCommand(params));

  return response;
}

/**
 * Update multiple fields in a map attribute in a single operation.
 *
 * This function demonstrates how to update multiple fields in a map
 * in a single DynamoDB operation.
 *
 * @param {Object} config - AWS configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {Object} key - The key of the item to update
 * @param {string} mapName - The name of the map attribute
 * @param {Object} updates - Object containing key-value pairs to update
 * @returns {Promise<Object>} - The response from DynamoDB
 */
async function updateMultipleMapFields(
  config,
  tableName,
```

```
  key,
  mapName,
  updates
) {
  // Initialize the DynamoDB client
  const client = new DynamoDBClient(config);
  const docClient = DynamoDBDocumentClient.from(client);

  // Build the update expression and expression attribute values
  let updateExpression = `SET ${mapName} = if_not_exists(${mapName}, :emptyMap)`;
  const expressionAttributeValues = {
    ":emptyMap": {}
  };

  // Add each update to the expression
  Object.entries(updates).forEach(([field, value], index) => {
    updateExpression += `, ${mapName}.${field} = :val${index}`;
    expressionAttributeValues[`:val${index}`] = value;
  });

  // Define the update parameters
  const params = {
    TableName: tableName,
    Key: key,
    UpdateExpression: updateExpression,
    ExpressionAttributeValues: expressionAttributeValues,
    ReturnValues: "UPDATED_NEW"
  };

  // Perform the update operation
  const response = await docClient.send(new UpdateCommand(params));

  return response;
}

/**
 * Get the current value of an item.
 *
 * Helper function to retrieve the current value of an item.
 *
 * @param {Object} config - AWS configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {Object} key - The key of the item to get
 * @returns {Promise<Object|null>} - The item or null if not found
```

```
 */
async function getItem(
  config,
  tableName,
  key
) {
  // Initialize the DynamoDB client
  const client = new DynamoDBClient(config);
  const docClient = DynamoDBDocumentClient.from(client);

  // Define the get parameters
  const params = {
    TableName: tableName,
    Key: key
  };

  // Perform the get operation
  const response = await docClient.send(new GetCommand(params));

  // Return the item if it exists, otherwise null
  return response.Item || null;
}

/**
 * Example of how to use the map attribute update functions.
 */
async function exampleUsage() {
  // Example parameters
  const config = { region: "us-west-2" };
  const tableName = "Users";
  const key = { UserId: "U12345" };

  console.log("Demonstrating different approaches to update map attributes in
 DynamoDB");

  try {
    // Example 1: Update a map attribute that might not exist (two-step approach)
    console.log("\nExample 1: Updating a map attribute that might not exist (two-
step approach)");
    const response1 = await updateMapAttributeSafe(
      config,
      tableName,
      key,
      "Preferences",
```

```
      "Theme",
      "Dark"
    );

    console.log("Updated preferences:", response1.Attributes);

    // Example 2: Update a map attribute using if_not_exists (elegant approach)
    console.log("\nExample 2: Updating a map attribute using if_not_exists
(elegant approach)");
    const response2 = await updateMapAttributeWithIfNotExists(
      config,
      tableName,
      key,
      "Settings",
      "NotificationsEnabled",
      true
    );

    console.log("Updated settings:", response2.Attributes);

    // Example 3: Update a deeply nested attribute
    console.log("\nExample 3: Updating a deeply nested attribute");
    const response3 = await addToNestedMap(
      config,
      tableName,
      key,
      ["Profile", "Address", "City"],
      "Seattle"
    );

    console.log("Updated nested attribute:", response3.Attributes);

    // Example 4: Update multiple fields in a map
    console.log("\nExample 4: Updating multiple fields in a map");
    const response4 = await updateMultipleMapFields(
      config,
      tableName,
      key,
      "ContactInfo",
      {
        Email: "user@example.com",
        Phone: "555-123-4567",
        PreferredContact: "Email"
      }
```

```
    );

    console.log("Updated multiple fields:", response4.Attributes);

    // Get the final state of the item
    console.log("\nFinal state of the item:");
    const item = await getItem(config, tableName, key);
    console.log(JSON.stringify(item, null, 2));

    // Explain the benefits of different approaches
    console.log("\nKey points about updating map attributes:");
    console.log("1. Use if_not_exists to handle maps that might not exist");
    console.log("2. Multiple updates can be combined in a single operation");
    console.log("3. Deeply nested attributes require creating parent maps");
    console.log("4. DynamoDB expressions are atomic - the entire update succeeds
 or fails");
    console.log("5. Using a single operation is more efficient than multiple
 separate updates");

  } catch (error) {
    console.error("Error:", error);
  }
}

// Export the functions
module.exports = {
  updateMapAttributeSafe,
  updateMapAttributeWithIfNotExists,
  addToNestedMap,
  updateMultipleMapFields,
  getItem,
  exampleUsage
};

// Run the example if this file is executed directly
if (require.main === module) {
  exampleUsage();
}
```

- For API details, see UpdateItem in *AWS SDK for JavaScript API Reference*.

Python

SDK for Python (Boto3)

Demonstrate map operations using AWS SDK for Python (Boto3).

```
"""
Example of updating map attributes in DynamoDB.

This module demonstrates how to update map attributes in DynamoDB, including
handling cases where the map attribute might not exist yet.
"""


import boto3
from typing import Any, Dict, Optional



def update_map_attribute_safe(
    table_name: str, key: Dict[str, Any], map_name: str, map_key: str, value: Any
) -> Dict[str, Any]:
    """
    Update a specific key in a map attribute, creating the map if it doesn't
 exist.

    This function demonstrates how to safely update a key within a map attribute,
    even if the map doesn't exist yet in the item.

    Args:
        table_name (str): The name of the DynamoDB table.
        key (Dict[str, Any]): The primary key of the item to update.
        map_name (str): The name of the map attribute.
        map_key (str): The key within the map to update.
        value (Any): The value to set for the map key.

    Returns:
        Dict[str, Any]: The response from DynamoDB containing the updated
 attribute values.
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)
```

```
    # Use SET with attribute_not_exists to safely update the map
    response = table.update_item(
        Key=key,
        UpdateExpression="SET #map.#key = :value",
        ExpressionAttributeNames={"#map": map_name, "#key": map_key},
        ExpressionAttributeValues={":value": value},
        ReturnValues="UPDATED_NEW",
    )

    return response


def add_to_nested_map(
    table_name: str, key: Dict[str, Any], path: str, value: Any
) -> Dict[str, Any]:
    """
    Add or update a value in a deeply nested map structure.

    This function demonstrates how to update a value at a specific path in a
    nested map structure, creating any intermediate maps as needed.

    Args:
        table_name (str): The name of the DynamoDB table.
        key (Dict[str, Any]): The primary key of the item to update.
        path (str): The path to the nested attribute (e.g.,
 "user.preferences.theme").
        value (Any): The value to set at the specified path.

    Returns:
        Dict[str, Any]: The response from DynamoDB containing the updated
 attribute values.
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # Split the path into components
    path_parts = path.split(".")

    # Build the update expression and attribute names
    update_expression = "SET "
    expression_attribute_names = {}
```

```
    # Build the path expression
    path_expression = ""
    for i, part in enumerate(path_parts):
        name_placeholder = f"#attr{i}"
        expression_attribute_names[name_placeholder] = part

        if i == 0:
            path_expression = name_placeholder
        else:
            path_expression += f".{name_placeholder}"

    # Complete the update expression
    update_expression += f"{path_expression} = :value"

    # Execute the update
    response = table.update_item(
        Key=key,
        UpdateExpression=update_expression,
        ExpressionAttributeNames=expression_attribute_names,
        ExpressionAttributeValues={":value": value},
        ReturnValues="UPDATED_NEW",
    )

    return response


def update_map_with_if_not_exists(
    table_name: str,
    key: Dict[str, Any],
    map_name: str,
    map_key: str,
    value: Any,
    default_map: Optional[Dict[str, Any]] = None,
) -> Dict[str, Any]:
    """
    Update a key in a map, creating the map with default values if it doesn't
 exist.

    This function demonstrates how to use if_not_exists to initialize a map with
    default values if it doesn't exist yet, and then update a specific key.

    Args:
        table_name (str): The name of the DynamoDB table.
        key (Dict[str, Any]): The primary key of the item to update.
```

```
        map_name (str): The name of the map attribute.
        map_key (str): The key within the map to update.
        value (Any): The value to set for the map key.
        default_map (Optional[Dict[str, Any]]): Default map values if the map
 doesn't exist.

    Returns:
        Dict[str, Any]: The response from DynamoDB containing the updated
 attribute values.
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # Set default map if not provided
    if default_map is None:
        default_map = {}

    # Create a map with the new key-value pair
    updated_map = default_map.copy()
    updated_map[map_key] = value

    # Use if_not_exists to initialize the map if it doesn't exist
    response = table.update_item(
        Key=key,
        UpdateExpression="SET #map = if_not_exists(#map, :default_map)",
        ExpressionAttributeNames={"#map": map_name},
        ExpressionAttributeValues={":default_map": updated_map},
        ReturnValues="UPDATED_NEW",
    )

    return response


def merge_into_map(
    table_name: str, key: Dict[str, Any], map_name: str, values_to_merge:
 Dict[str, Any]
) -> Dict[str, Any]:
    """
    Merge multiple key-value pairs into a map attribute.

    This function demonstrates how to update multiple keys in a map attribute
    in a single operation, without overwriting the entire map.
```

```
    Args:
        table_name (str): The name of the DynamoDB table.
        key (Dict[str, Any]): The primary key of the item to update.
        map_name (str): The name of the map attribute.
        values_to_merge (Dict[str, Any]): Key-value pairs to merge into the map.

    Returns:
        Dict[str, Any]: The response from DynamoDB containing the updated
attribute values.
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # Build the update expression for each key-value pair
    update_expression = "SET "
    expression_attribute_names = {"#map": map_name}
    expression_attribute_values = {}

    # Add each key-value pair to the update expression
    for i, (k, v) in enumerate(values_to_merge.items()):
        key_placeholder = f"#key{i}"
        value_placeholder = f":value{i}"

        expression_attribute_names[key_placeholder] = k
        expression_attribute_values[value_placeholder] = v

        if i > 0:
            update_expression += ", "
        update_expression += f"#map.{key_placeholder} = {value_placeholder}"

    # Execute the update
    response = table.update_item(
        Key=key,
        UpdateExpression=update_expression,
        ExpressionAttributeNames=expression_attribute_names,
        ExpressionAttributeValues=expression_attribute_values,
        ReturnValues="UPDATED_NEW",
    )

    return response
```

```
def example_usage():
    """Example of how to use the map attribute update functions."""
    # Example parameters
    table_name = "UserProfiles"
    key = {"UserId": "user123"}

    print("Example 1: Updating a specific key in a map attribute")
    try:
        response = update_map_attribute_safe(
            table_name=table_name, key=key, map_name="Preferences",
 map_key="Theme", value="Dark"
        )
        print(f"Map attribute updated successfully: {response.get('Attributes',
 {})}")
    except Exception as e:
        print(f"Error updating map attribute: {e}")

    print("\nExample 2: Adding a value to a deeply nested map")
    try:
        response = add_to_nested_map(
            table_name=table_name, key=key, path="Settings.Notifications.Email",
 value=True
        )
        print(f"Nested map updated successfully: {response.get('Attributes',
 {})}")
    except Exception as e:
        print(f"Error updating nested map: {e}")

    print("\nExample 3: Initializing a map with default values if it doesn't
 exist")
    try:
        default_map = {"Language": "English", "Currency": "USD", "TimeZone":
 "UTC"}

        response = update_map_with_if_not_exists(
            table_name=table_name,
            key={"UserId": "newuser456"},
            map_name="Preferences",
            map_key="Theme",
            value="Light",
            default_map=default_map,
        )
        print(f"Map initialized with defaults: {response.get('Attributes', {})}")
```

```
    except Exception as e:
        print(f"Error initializing map: {e}")

    print("\nExample 4: Merging multiple values into a map")
    try:
        values_to_merge = {
            "NotificationsEnabled": True,
            "EmailFrequency": "Daily",
            "PushNotifications": False,
        }

        response = merge_into_map(
            table_name=table_name,
            key=key,
            map_name="NotificationSettings",
            values_to_merge=values_to_merge,
        )
        print(f"Multiple values merged into map: {response.get('Attributes',
 {})}")
    except Exception as e:
        print(f"Error merging values into map: {e}")

    print("\nBest practices for working with map attributes in DynamoDB:")
    print("1. Use dot notation to access and update nested attributes")
    print("2. Use ExpressionAttributeNames to handle reserved words and special
 characters")
    print("3. Use if_not_exists() to handle cases where attributes might not
 exist")
    print("4. Update specific map keys rather than overwriting the entire map")
    print("5. Use a single update operation to modify multiple map keys for
 better performance")
    print("6. Consider your data model carefully to minimize the need for deeply
 nested attributes")



if __name__ == "__main__":
    example_usage()
```

- For API details, see UpdateItem in *AWS SDK for Python (Boto3) API Reference*.

For a complete list of AWS SDK developer guides and code examples, see Using DynamoDB with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Perform set operations in DynamoDB with an AWS SDK

The following code examples show how to perform set operations in DynamoDB.

- Add elements to a set attribute.
- Remove elements from a set attribute.
- Use ADD and DELETE operations with sets.

Java

SDK for Java 2.x

Demonstrate set operations using AWS SDK for Java 2.x.

```
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.GetItemRequest;
import software.amazon.awssdk.services.dynamodb.model.GetItemResponse;
import software.amazon.awssdk.services.dynamodb.model.ReturnValue;
import software.amazon.awssdk.services.dynamodb.model.UpdateItemRequest;
import software.amazon.awssdk.services.dynamodb.model.UpdateItemResponse;

import java.util.HashMap;
import java.util.HashSet;
import java.util.Map;
import java.util.Set;

    /**
     * Adds values to a string set attribute.
     *
     * <p>This method demonstrates how to use the ADD operation to add values
     * to a string set attribute.
     *
     * @param dynamoDbClient The DynamoDB client
     * @param tableName The name of the DynamoDB table
     * @param key The key of the item to update
     * @param setAttributeName The name of the set attribute
```

```
     * @param valuesToAdd The values to add to the set
     * @return The response from DynamoDB
     * @throws DynamoDbException if an error occurs during the operation
     */
    public static UpdateItemResponse addToStringSet(
        DynamoDbClient dynamoDbClient,
        String tableName,
        Map<String, AttributeValue> key,
        String setAttributeName,
        Set<String> valuesToAdd) {

        // Create a string set value from the values to add
        AttributeValue setValue =
AttributeValue.builder().ss(valuesToAdd).build();

        // Define the update parameters
        UpdateItemRequest request = UpdateItemRequest.builder()
            .tableName(tableName)
            .key(key)
            .updateExpression("ADD #setAttr :valuesToAdd")
            .expressionAttributeNames(Map.of("#setAttr", setAttributeName))
            .expressionAttributeValues(Map.of(":valuesToAdd", setValue))
            .returnValues(ReturnValue.UPDATED_NEW)
            .build();

        // Perform the update operation
        return dynamoDbClient.updateItem(request);
    }

    /**
     * Adds values to a number set attribute.
     *
     * <p>This method demonstrates how to use the ADD operation to add values
     * to a number set attribute.
     *
     * @param dynamoDbClient The DynamoDB client
     * @param tableName The name of the DynamoDB table
     * @param key The key of the item to update
     * @param setAttributeName The name of the set attribute
     * @param valuesToAdd The values to add to the set
     * @return The response from DynamoDB
     * @throws DynamoDbException if an error occurs during the operation
     */
    public static UpdateItemResponse addToNumberSet(
```

```
        DynamoDbClient dynamoDbClient,
        String tableName,
        Map<String, AttributeValue> key,
        String setAttributeName,
        Set<Number> valuesToAdd) {

        // Convert numbers to strings for DynamoDB
        Set<String> stringValues = new HashSet<>();
        for (Number value : valuesToAdd) {
            stringValues.add(value.toString());
        }

        // Create a number set value from the values to add
        AttributeValue setValue =
AttributeValue.builder().ns(stringValues).build();

        // Define the update parameters
        UpdateItemRequest request = UpdateItemRequest.builder()
            .tableName(tableName)
            .key(key)
            .updateExpression("ADD #setAttr :valuesToAdd")
            .expressionAttributeNames(Map.of("#setAttr", setAttributeName))
            .expressionAttributeValues(Map.of(":valuesToAdd", setValue))
            .returnValues(ReturnValue.UPDATED_NEW)
            .build();

        // Perform the update operation
        return dynamoDbClient.updateItem(request);
    }

    /**
     * Removes values from a set attribute.
     *
     * <p>This method demonstrates how to use the DELETE operation to remove
values
     * from a set attribute.
     *
     * @param dynamoDbClient The DynamoDB client
     * @param tableName The name of the DynamoDB table
     * @param key The key of the item to update
     * @param setAttributeName The name of the set attribute
     * @param valuesToRemove The values to remove from the set
     * @param isNumberSet Whether the set is a number set (true) or string set
(false)
```

```
     * @return The response from DynamoDB
     * @throws DynamoDbException if an error occurs during the operation
     */
    public static UpdateItemResponse removeFromSet(
        DynamoDbClient dynamoDbClient,
        String tableName,
        Map<String, AttributeValue> key,
        String setAttributeName,
        Set<?> valuesToRemove,
        boolean isNumberSet) {

        AttributeValue setValue;

        if (isNumberSet) {
            // Convert numbers to strings for DynamoDB
            Set<String> stringValues = new HashSet<>();
            for (Object value : valuesToRemove) {
                if (value instanceof Number) {
                    stringValues.add(value.toString());
                } else {
                    throw new IllegalArgumentException("Values must be numbers
for a number set");
                }
            }

            setValue = AttributeValue.builder().ns(stringValues).build();
        } else {
            // Convert objects to strings for DynamoDB
            Set<String> stringValues = new HashSet<>();
            for (Object value : valuesToRemove) {
                stringValues.add(value.toString());
            }

            setValue = AttributeValue.builder().ss(stringValues).build();
        }

        // Define the update parameters
        UpdateItemRequest request = UpdateItemRequest.builder()
            .tableName(tableName)
            .key(key)
            .updateExpression("DELETE #setAttr :valuesToRemove")
            .expressionAttributeNames(Map.of("#setAttr", setAttributeName))
            .expressionAttributeValues(Map.of(":valuesToRemove", setValue))
            .returnValues(ReturnValue.UPDATED_NEW)
```

```
            .build();

        // Perform the update operation
        return dynamoDbClient.updateItem(request);
    }

    /**
     * Checks if a value exists in a set attribute.
     *
     * <p>This method demonstrates how to use the contains function to check
     * if a value exists in a set attribute.
     *
     * @param dynamoDbClient The DynamoDB client
     * @param tableName The name of the DynamoDB table
     * @param key The key of the item to check
     * @param setAttributeName The name of the set attribute
     * @param valueToCheck The value to check for
     * @return Map containing the result of the check
     * @throws DynamoDbException if an error occurs during the operation
     */
    public static Map<String, Object> checkIfValueInSet(
        DynamoDbClient dynamoDbClient,
        String tableName,
        Map<String, AttributeValue> key,
        String setAttributeName,
        String valueToCheck) {

        Map<String, Object> result = new HashMap<>();

        try {
            // Define the update parameters with a condition expression
            UpdateItemRequest request = UpdateItemRequest.builder()
                .tableName(tableName)
                .key(key)
                .updateExpression("SET #tempAttr = :tempVal")
                .conditionExpression("contains(#setAttr, :valueToCheck)")
                .expressionAttributeNames(Map.of("#setAttr", setAttributeName,
"#tempAttr", "TempAttribute"))
                .expressionAttributeValues(Map.of(
                    ":valueToCheck",
AttributeValue.builder().s(valueToCheck).build(),
                    ":tempVal", AttributeValue.builder().s("TempValue").build()))
                .returnValues(ReturnValue.UPDATED_NEW)
                .build();
```

```
            // Attempt the update operation
            dynamoDbClient.updateItem(request);

            // If we get here, the condition was met
            result.put("exists", true);
            result.put("message", "Value '" + valueToCheck + "' exists in the
set");

            // Clean up the temporary attribute
            UpdateItemRequest cleanupRequest = UpdateItemRequest.builder()
                .tableName(tableName)
                .key(key)
                .updateExpression("REMOVE #tempAttr")
                .expressionAttributeNames(Map.of("#tempAttr", "TempAttribute"))
                .build();

            dynamoDbClient.updateItem(cleanupRequest);

        } catch (DynamoDbException e) {
            if (e.getMessage().contains("ConditionalCheckFailed")) {
                // The condition was not met
                result.put("exists", false);
                result.put("message", "Value '" + valueToCheck + "' does not
exist in the set");
            } else {
                // Some other error occurred
                result.put("exists", false);
                result.put("message", "Error checking set: " + e.getMessage());
                result.put("error", e.getClass().getSimpleName());
            }
        }

        return result;
    }

    /**
     * Creates a set with multiple values in a single operation.
     *
     * <p>This method demonstrates how to create a set with multiple values
     * in a single update operation.
     *
     * @param dynamoDbClient The DynamoDB client
     * @param tableName The name of the DynamoDB table
```

```
     * @param key The key of the item to update
     * @param setAttributeName The name of the set attribute
     * @param setValues The values to include in the set
     * @param isNumberSet Whether to create a number set (true) or string set
(false)
     * @return The response from DynamoDB
     * @throws DynamoDbException if an error occurs during the operation
     */
    public static UpdateItemResponse createSetWithValues(
        DynamoDbClient dynamoDbClient,
        String tableName,
        Map<String, AttributeValue> key,
        String setAttributeName,
        Set<?> setValues,
        boolean isNumberSet) {

        AttributeValue setValue;

        if (isNumberSet) {
            // Convert numbers to strings for DynamoDB
            Set<String> stringValues = new HashSet<>();
            for (Object value : setValues) {
                if (value instanceof Number) {
                    stringValues.add(value.toString());
                } else {
                    throw new IllegalArgumentException("Values must be numbers
for a number set");
                }
            }

            setValue = AttributeValue.builder().ns(stringValues).build();
        } else {
            // Convert objects to strings for DynamoDB
            Set<String> stringValues = new HashSet<>();
            for (Object value : setValues) {
                stringValues.add(value.toString());
            }

            setValue = AttributeValue.builder().ss(stringValues).build();
        }

        // Define the update parameters
        UpdateItemRequest request = UpdateItemRequest.builder()
            .tableName(tableName)
```

```
            .key(key)
            .updateExpression("SET #setAttr = :setValue")
            .expressionAttributeNames(Map.of("#setAttr", setAttributeName))
            .expressionAttributeValues(Map.of(":setValue", setValue))
            .returnValues(ReturnValue.UPDATED_NEW)
            .build();

        // Perform the update operation
        return dynamoDbClient.updateItem(request);
    }

    /**
     * Gets the current value of a set attribute.
     *
     * <p>Helper method to retrieve the current value of a set attribute.
     *
     * @param dynamoDbClient The DynamoDB client
     * @param tableName The name of the DynamoDB table
     * @param key The key of the item to get
     * @param setAttributeName The name of the set attribute
     * @return The set attribute value or null if not found
     * @throws DynamoDbException if an error occurs during the operation
     */
    public static AttributeValue getSetAttribute(
        DynamoDbClient dynamoDbClient, String tableName, Map<String,
AttributeValue> key, String setAttributeName) {

        // Define the get parameters
        GetItemRequest request = GetItemRequest.builder()
            .tableName(tableName)
            .key(key)
            .projectionExpression(setAttributeName)
            .build();

        try {
            // Perform the get operation
            GetItemResponse response = dynamoDbClient.getItem(request);

            // Return the set attribute if it exists, otherwise null
            if (response.item() != null &&
response.item().containsKey(setAttributeName)) {
                return response.item().get(setAttributeName);
            }
```

```
            return null;
        } catch (DynamoDbException e) {
            throw DynamoDbException.builder()
                .message("Failed to get set attribute: " + e.getMessage())
                .cause(e)
                .build();
        }
    }
```

Example usage of set operations with AWS SDK for Java 2.x.

```
    public static void exampleUsage(DynamoDbClient dynamoDbClient, String
tableName) {
        // Example key
        Map<String, AttributeValue> key = new HashMap<>();
        key.put("ProductId", AttributeValue.builder().s("P12345").build());

        System.out.println("Demonstrating set operations in DynamoDB");

        try {
            // Example 1: Create a string set with multiple values
            System.out.println("\nExample 1: Creating a string set with multiple
values");
            Set<String> tags = new HashSet<>();
            tags.add("Electronics");
            tags.add("Gadget");
            tags.add("Smartphone");

            UpdateItemResponse createResponse = createSetWithValues(
                dynamoDbClient, tableName, key, "Tags", tags, false // Not a
number set
                );

            System.out.println("Created set attribute: " +
createResponse.attributes());

            // Example 2: Add values to a string set
            System.out.println("\nExample 2: Adding values to a string set");
            Set<String> additionalTags = new HashSet<>();
            additionalTags.add("Mobile");
            additionalTags.add("Wireless");
```

```
            UpdateItemResponse addResponse = addToStringSet(dynamoDbClient,
tableName, key, "Tags", additionalTags);

            System.out.println("Updated set attribute: " +
addResponse.attributes());

            // Example 3: Create a number set with multiple values
            System.out.println("\nExample 3: Creating a number set with multiple
values");
            Set<Number> ratings = new HashSet<>();
            ratings.add(4);
            ratings.add(5);
            ratings.add(4.5);

            UpdateItemResponse createNumberSetResponse = createSetWithValues(
                dynamoDbClient, tableName, key, "Ratings", ratings, true // Is a
number set
                );

            System.out.println("Created number set attribute: " +
createNumberSetResponse.attributes());

            // Example 4: Add values to a number set
            System.out.println("\nExample 4: Adding values to a number set");
            Set<Number> additionalRatings = new HashSet<>();
            additionalRatings.add(3.5);
            additionalRatings.add(4.2);

            UpdateItemResponse addNumberResponse =
                addToNumberSet(dynamoDbClient, tableName, key, "Ratings",
additionalRatings);

            System.out.println("Updated number set attribute: " +
addNumberResponse.attributes());

            // Example 5: Remove values from a set
            System.out.println("\nExample 5: Removing values from a set");
            Set<String> tagsToRemove = new HashSet<>();
            tagsToRemove.add("Gadget");

            UpdateItemResponse removeResponse = removeFromSet(
                dynamoDbClient, tableName, key, "Tags", tagsToRemove, false //
Not a number set
                );
```

```
            System.out.println("Updated set after removal: " +
removeResponse.attributes());

            // Example 6: Check if a value exists in a set
            System.out.println("\nExample 6: Checking if a value exists in a
set");
            Map<String, Object> checkResult = checkIfValueInSet(dynamoDbClient,
tableName, key, "Tags", "Electronics");

            System.out.println("Check result: " + checkResult.get("message"));

            // Example 7: Get the current value of a set attribute
            System.out.println("\nExample 7: Getting the current value of a set
attribute");
            AttributeValue currentStringSet = getSetAttribute(dynamoDbClient,
tableName, key, "Tags");

            if (currentStringSet != null && currentStringSet.ss() != null) {
                System.out.println("Current string set values: " +
currentStringSet.ss());
            } else {
                System.out.println("String set attribute not found");
            }

            AttributeValue currentNumberSet = getSetAttribute(dynamoDbClient,
tableName, key, "Ratings");

            if (currentNumberSet != null && currentNumberSet.ns() != null) {
                System.out.println("Current number set values: " +
currentNumberSet.ns());
            } else {
                System.out.println("Number set attribute not found");
            }

            // Explain set operations
            System.out.println("\nKey points about DynamoDB set operations:");
            System.out.println(
                "1. DynamoDB supports three set types: string sets (SS), number
sets (NS), and binary sets (BS)");
            System.out.println("2. Sets can only contain elements of the same
type");
            System.out.println("3. Use ADD to add elements to a set");
            System.out.println("4. Use DELETE to remove elements from a set");
```

```
            System.out.println("5. Sets automatically remove duplicate values");
            System.out.println("6. Sets are unordered collections");
            System.out.println("7. Use the contains function to check if a value
 exists in a set");
            System.out.println("8. You can create a set with multiple values in a
 single operation");

        } catch (DynamoDbException e) {
            System.err.println("Error: " + e.getMessage());
            e.printStackTrace();
        }
    }
```

- For API details, see UpdateItem in *AWS SDK for Java 2.x API Reference*.

JavaScript

SDK for JavaScript (v3)

Demonstrate set operations using AWS SDK for JavaScript.

```
const { DynamoDBClient } = require("@aws-sdk/client-dynamodb");
const {
  DynamoDBDocumentClient,
  UpdateCommand,
  GetCommand
} = require("@aws-sdk/lib-dynamodb");

/**
 * Add elements to a set attribute.
 *
 * This function demonstrates using the ADD operation to add elements to a set.
 *
 * @param {Object} config - AWS configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {Object} key - The key of the item to update
 * @param {string} setName - The name of the set attribute
 * @param {Array} values - The values to add to the set
 * @param {string} setType - The type of set ('string', 'number', or 'binary')
 * @returns {Promise<Object>} - The response from DynamoDB
 */
async function addToSet(
```

```
  config,
  tableName,
  key,
  setName,
  values,
  setType = 'string'
) {
  // Initialize the DynamoDB client
  const client = new DynamoDBClient(config);
  const docClient = DynamoDBDocumentClient.from(client);

  // Create the appropriate set type
  let setValues;
  if (setType === 'string') {
    setValues = new Set(values.map(String));
  } else if (setType === 'number') {
    setValues = new Set(values.map(Number));
  } else if (setType === 'binary') {
    setValues = new Set(values);
  } else {
    throw new Error(`Unsupported set type: ${setType}`);
  }

  // Define the update parameters using ADD
  const params = {
    TableName: tableName,
    Key: key,
    UpdateExpression: `ADD ${setName} :values`,
    ExpressionAttributeValues: {
      ":values": setValues
    },
    ReturnValues: "UPDATED_NEW"
  };

  // Perform the update operation
  const response = await docClient.send(new UpdateCommand(params));

  return response;
}

/**
 * Remove elements from a set attribute.
 *
```

```
 * This function demonstrates using the DELETE operation to remove elements from
 a set.
 *
 * @param {Object} config - AWS configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {Object} key - The key of the item to update
 * @param {string} setName - The name of the set attribute
 * @param {Array} values - The values to remove from the set
 * @param {string} setType - The type of set ('string', 'number', or 'binary')
 * @returns {Promise<Object>} - The response from DynamoDB
 */
async function removeFromSet(
  config,
  tableName,
  key,
  setName,
  values,
  setType = 'string'
) {
  // Initialize the DynamoDB client
  const client = new DynamoDBClient(config);
  const docClient = DynamoDBDocumentClient.from(client);

  // Create the appropriate set type
  let setValues;
  if (setType === 'string') {
    setValues = new Set(values.map(String));
  } else if (setType === 'number') {
    setValues = new Set(values.map(Number));
  } else if (setType === 'binary') {
    setValues = new Set(values);
  } else {
    throw new Error(`Unsupported set type: ${setType}`);
  }

  // Define the update parameters using DELETE
  const params = {
    TableName: tableName,
    Key: key,
    UpdateExpression: `DELETE ${setName} :values`,
    ExpressionAttributeValues: {
      ":values": setValues
    },
    ReturnValues: "UPDATED_NEW"
```

```
  };

  // Perform the update operation
  const response = await docClient.send(new UpdateCommand(params));

  return response;
}

/**
 * Create a new set attribute with initial values.
 *
 * This function demonstrates using the SET operation to create a new set
 attribute.
 *
 * @param {Object} config - AWS configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {Object} key - The key of the item to update
 * @param {string} setName - The name of the set attribute
 * @param {Array} values - The initial values for the set
 * @param {string} setType - The type of set ('string', 'number', or 'binary')
 * @returns {Promise<Object>} - The response from DynamoDB
 */
async function createSet(
  config,
  tableName,
  key,
  setName,
  values,
  setType = 'string'
) {
  // Initialize the DynamoDB client
  const client = new DynamoDBClient(config);
  const docClient = DynamoDBDocumentClient.from(client);

  // Create the appropriate set type
  let setValues;
  if (setType === 'string') {
    setValues = new Set(values.map(String));
  } else if (setType === 'number') {
    setValues = new Set(values.map(Number));
  } else if (setType === 'binary') {
    setValues = new Set(values);
  } else {
    throw new Error(`Unsupported set type: ${setType}`);
```

```
  }

  // Define the update parameters using SET
  const params = {
    TableName: tableName,
    Key: key,
    UpdateExpression: `SET ${setName} = :values`,
    ExpressionAttributeValues: {
      ":values": setValues
    },
    ReturnValues: "UPDATED_NEW"
  };

  // Perform the update operation
  const response = await docClient.send(new UpdateCommand(params));

  return response;
}

/**
 * Replace an entire set attribute with a new set of values.
 *
 * This function demonstrates using the SET operation to replace an entire set.
 *
 * @param {Object} config - AWS configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {Object} key - The key of the item to update
 * @param {string} setName - The name of the set attribute
 * @param {Array} values - The new values for the set
 * @param {string} setType - The type of set ('string', 'number', or 'binary')
 * @returns {Promise<Object>} - The response from DynamoDB
 */
async function replaceSet(
  config,
  tableName,
  key,
  setName,
  values,
  setType = 'string'
) {
  // This is the same as createSet, but included for clarity of intent
  return await createSet(config, tableName, key, setName, values, setType);
}
```

```
/**
 * Remove the last element from a set and handle the empty set case.
 *
 * This function demonstrates what happens when you delete the last element of a
 set.
 *
 * @param {Object} config - AWS configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {Object} key - The key of the item to update
 * @param {string} setName - The name of the set attribute
 * @returns {Promise<Object>} - The result of the operation
 */
async function removeLastElementFromSet(
  config,
  tableName,
  key,
  setName
) {
  // Initialize the DynamoDB client
  const client = new DynamoDBClient(config);
  const docClient = DynamoDBDocumentClient.from(client);

  // First, get the current item to check the set
  const currentItem = await getItem(config, tableName, key);

  // Check if the set exists and has elements
  if (!currentItem || !currentItem[setName] || currentItem[setName].size === 0) {
    return {
      success: false,
      message: "Set doesn't exist or is already empty",
      item: currentItem
    };
  }

  // Get the set values
  const setValues = Array.from(currentItem[setName]);

  // If there's only one element left, remove the attribute entirely
  if (setValues.length === 1) {
    // Define the update parameters to remove the attribute
    const params = {
      TableName: tableName,
      Key: key,
      UpdateExpression: `REMOVE ${setName}`,
```

```
      ReturnValues: "UPDATED_NEW"
    };

    // Perform the update operation
    await docClient.send(new UpdateCommand(params));

    return {
      success: true,
      message: "Last element removed, attribute has been deleted",
      removedValue: setValues[0]
    };
  } else {
    // Otherwise, remove just the last element
    // Create a set with just the last element
    const lastElement = setValues[setValues.length - 1];
    const setType = typeof lastElement === 'number' ? 'number' : 'string';

    // Remove the last element
    const response = await removeFromSet(
      config,
      tableName,
      key,
      setName,
      [lastElement],
      setType
    );

    return {
      success: true,
      message: "Last element removed, set still contains elements",
      removedValue: lastElement,
      remainingSet: response.Attributes[setName]
    };
  }
}

/**
 * Get the current value of an item.
 *
 * Helper function to retrieve the current value of an item.
 *
 * @param {Object} config - AWS configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {Object} key - The key of the item to get
```

```
 * @returns {Promise<Object|null>} - The item or null if not found
 */
async function getItem(
  config,
  tableName,
  key
) {
  // Initialize the DynamoDB client
  const client = new DynamoDBClient(config);
  const docClient = DynamoDBDocumentClient.from(client);

  // Define the get parameters
  const params = {
    TableName: tableName,
    Key: key
  };

  // Perform the get operation
  const response = await docClient.send(new GetCommand(params));

  // Return the item if it exists, otherwise null
  return response.Item || null;
}
```

Example usage of set operations with AWS SDK for JavaScript.

```
/**
 * Example of how to work with sets in DynamoDB.
 */
async function exampleUsage() {
  // Example parameters
  const config = { region: "us-west-2" };
  const tableName = "Users";
  const key = { UserId: "U12345" };

  console.log("Demonstrating set operations in DynamoDB");

  try {
    // Example 1: Create a string set
    console.log("\nExample 1: Creating a string set");
    const response1 = await createSet(
      config,
```

```
      tableName,
      key,
      "Interests",
      ["Reading", "Hiking", "Cooking"],
      "string"
    );

    console.log("Created set:", response1.Attributes);

    // Example 2: Add elements to a set
    console.log("\nExample 2: Adding elements to a set");
    const response2 = await addToSet(
      config,
      tableName,
      key,
      "Interests",
      ["Photography", "Travel"],
      "string"
    );

    console.log("Updated set after adding elements:", response2.Attributes);

    // Example 3: Remove elements from a set
    console.log("\nExample 3: Removing elements from a set");
    const response3 = await removeFromSet(
      config,
      tableName,
      key,
      "Interests",
      ["Cooking"],
      "string"
    );

    console.log("Updated set after removing elements:", response3.Attributes);

    // Example 4: Create a number set
    console.log("\nExample 4: Creating a number set");
    const response4 = await createSet(
      config,
      tableName,
      key,
      "FavoriteNumbers",
      [7, 42, 99],
      "number"
```

```
    );

    console.log("Created number set:", response4.Attributes);

    // Example 5: Replace an entire set
    console.log("\nExample 5: Replacing an entire set");
    const response5 = await replaceSet(
      config,
      tableName,
      key,
      "Interests",
      ["Gaming", "Movies", "Music"],
      "string"
    );

    console.log("Replaced set:", response5.Attributes);

    // Example 6: Remove the last element from a set
    console.log("\nExample 6: Removing the last element from a set");

    // First, create a set with just one element
    await createSet(
      config,
      tableName,
      { UserId: "U67890" },
      "Tags",
      ["LastTag"],
      "string"
    );

    // Then, remove the last element
    const response6 = await removeLastElementFromSet(
      config,
      tableName,
      { UserId: "U67890" },
      "Tags"
    );

    console.log(response6.message);
    console.log("Removed value:", response6.removedValue);

    // Get the final state of the items
    console.log("\nFinal state of the items:");
    const item1 = await getItem(config, tableName, key);
```

```
    console.log("User U12345:", JSON.stringify(item1, null, 2));

    const item2 = await getItem(config, tableName, { UserId: "U67890" });
    console.log("User U67890:", JSON.stringify(item2, null, 2));

    // Explain set operations
    console.log("\nKey points about set operations in DynamoDB:");
    console.log("1. Use ADD to add elements to a set (duplicates are
 automatically removed)");
    console.log("2. Use DELETE to remove elements from a set");
    console.log("3. Use SET to create a new set or replace an existing one");
    console.log("4. DynamoDB supports three types of sets: string sets, number
 sets, and binary sets");
    console.log("5. When you delete the last element from a set, the attribute
 remains as an empty set");
    console.log("6. To remove an empty set, use the REMOVE operation");
    console.log("7. Sets automatically maintain unique values (no duplicates)");
    console.log("8. You cannot mix data types within a set");

  } catch (error) {
    console.error("Error:", error);
  }
}
```

- For API details, see UpdateItem in *AWS SDK for JavaScript API Reference*.

Python

SDK for Python (Boto3)

Demonstrate set operations using AWS SDK for Python (Boto3).

```
import boto3
from typing import Any, Dict, List


def create_set_attribute(
    table_name: str,
    key: Dict[str, Any],
    set_name: str,
    set_values: List[Any],
    set_type: str = "string",
```

```
) -> Dict[str, Any]:
    """
    Create a new set attribute or add elements to an existing set.

    This function demonstrates how to use the ADD operation to create a new set
    or add elements to an existing set.

    Args:
        table_name (str): The name of the DynamoDB table.
        key (Dict[str, Any]): The primary key of the item to update.
        set_name (str): The name of the set attribute.
        set_values (List[Any]): The values to add to the set.
        set_type (str, optional): The type of set to create: "string", "number",
 or "binary".
            Defaults to "string".

    Returns:
        Dict[str, Any]: The response from DynamoDB containing the updated
 attribute values.
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # Convert the list to a DynamoDB set based on the specified type
    if set_type == "string":
        dynamo_set = set(str(value) for value in set_values)
    elif set_type == "number":
        # We need to use actual float values for the DynamoDB API
        # but mypy expects strings in sets, so we need to use type: ignore
        dynamo_set = set(float(value) for value in set_values)  # type: ignore
    else:  # binary set is not directly supported in high-level API, handled
 differently
        raise ValueError("Binary sets are not supported in this example")

    # Use the ADD operation to create or update the set
    response = table.update_item(
        Key=key,
        UpdateExpression="ADD #set_attr :set_values",
        ExpressionAttributeNames={"#set_attr": set_name},
        ExpressionAttributeValues={":set_values": dynamo_set},
        ReturnValues="UPDATED_NEW",
    )
```

```
    return response


def add_to_set(
    table_name: str, key: Dict[str, Any], set_name: str, values_to_add: List[Any]
) -> Dict[str, Any]:
    """
    Add elements to an existing set attribute.

    This function demonstrates how to use the ADD operation to add elements to an
 existing set.
    If the set doesn't exist, it will be created.

    Args:
        table_name (str): The name of the DynamoDB table.
        key (Dict[str, Any]): The primary key of the item to update.
        set_name (str): The name of the set attribute.
        values_to_add (List[Any]): The values to add to the set.

    Returns:
        Dict[str, Any]: The response from DynamoDB containing the updated
 attribute values.
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # Convert the list to a set (assuming string set for simplicity)
    dynamo_set = set(str(value) for value in values_to_add)

    # Use the ADD operation to add values to the set
    response = table.update_item(
        Key=key,
        UpdateExpression="ADD #set_attr :values_to_add",
        ExpressionAttributeNames={"#set_attr": set_name},
        ExpressionAttributeValues={":values_to_add": dynamo_set},
        ReturnValues="UPDATED_NEW",
    )

    return response


def remove_from_set(
```

```
    table_name: str, key: Dict[str, Any], set_name: str, values_to_remove:
 List[Any]
) -> Dict[str, Any]:
    """
    Remove elements from a set attribute.

    This function demonstrates how to use the DELETE operation to remove elements
 from a set.
    If the last element is removed, the attribute will be deleted entirely.

    Args:
        table_name (str): The name of the DynamoDB table.
        key (Dict[str, Any]): The primary key of the item to update.
        set_name (str): The name of the set attribute.
        values_to_remove (List[Any]): The values to remove from the set.

    Returns:
        Dict[str, Any]: The response from DynamoDB containing the updated
 attribute values.
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # Convert the list to a set (assuming string set for simplicity)
    dynamo_set = set(str(value) for value in values_to_remove)

    # Use the DELETE operation to remove values from the set
    response = table.update_item(
        Key=key,
        UpdateExpression="DELETE #set_attr :values_to_remove",
        ExpressionAttributeNames={"#set_attr": set_name},
        ExpressionAttributeValues={":values_to_remove": dynamo_set},
        ReturnValues="UPDATED_NEW",
    )

    return response


def check_if_set_exists(table_name: str, key: Dict[str, Any], set_name: str) ->
 bool:
    """
    Check if a set attribute exists in an item.
```

```
    This function demonstrates how to check if a set attribute exists after
    potentially removing all elements from it.

    Args:
        table_name (str): The name of the DynamoDB table.
        key (Dict[str, Any]): The primary key of the item to check.
        set_name (str): The name of the set attribute.

    Returns:
        bool: True if the set attribute exists, False otherwise.
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # Get the item
    response = table.get_item(
        Key=key, ProjectionExpression="#set_attr",
 ExpressionAttributeNames={"#set_attr": set_name}
    )

    # Check if the item exists and has the set attribute
    return "Item" in response and set_name in response["Item"]


def demonstrate_last_element_removal(
    table_name: str, key: Dict[str, Any], set_name: str
) -> Dict[str, Any]:
    """
    Demonstrate what happens when you remove the last element from a set.

    This function creates a set with a single element, then removes that element,
    showing that the attribute is completely removed when the last element is
 deleted.

    Args:
        table_name (str): The name of the DynamoDB table.
        key (Dict[str, Any]): The primary key of the item to update.
        set_name (str): The name of the set attribute.

    Returns:
        Dict[str, Any]: A dictionary containing the results of the demonstration.
    """
    # Step 1: Create a set with a single element
```

```
    create_response = create_set_attribute(
        table_name=table_name,
        key=key,
        set_name=set_name,
        set_values=["last_element"],
        set_type="string",
    )

    # Step 2: Check that the set exists
    exists_before = check_if_set_exists(table_name, key, set_name)

    # Step 3: Remove the last element
    delete_response = remove_from_set(
        table_name=table_name, key=key, set_name=set_name,
 values_to_remove=["last_element"]
    )

    # Step 4: Check if the set still exists
    exists_after = check_if_set_exists(table_name, key, set_name)

    # Return the results
    return {
        "create_response": create_response,
        "exists_before": exists_before,
        "delete_response": delete_response,
        "exists_after": exists_after,
    }


def work_with_number_set(
    table_name: str,
    key: Dict[str, Any],
    set_name: str,
    initial_values: List[float],
    values_to_add: List[float],
    values_to_remove: List[float],
) -> Dict[str, Any]:
    """
    Demonstrate working with a number set in DynamoDB.

    This function shows how to create and manipulate a set of numbers.

    Args:
        table_name (str): The name of the DynamoDB table.
```

```
        key (Dict[str, Any]): The primary key of the item to update.
        set_name (str): The name of the set attribute.
        initial_values (List[float]): The initial values for the set.
        values_to_add (List[float]): Values to add to the set.
        values_to_remove (List[float]): Values to remove from the set.

    Returns:
        Dict[str, Any]: A dictionary containing the responses from each
operation.
    """
    # Step 1: Create the number set
    create_response = create_set_attribute(
        table_name=table_name,
        key=key,
        set_name=set_name,
        set_values=initial_values,
        set_type="number",
    )

    # Step 2: Add more numbers to the set
    add_response = add_to_set(
        table_name=table_name, key=key, set_name=set_name,
values_to_add=values_to_add
    )

    # Step 3: Remove some numbers from the set
    remove_response = remove_from_set(
        table_name=table_name, key=key, set_name=set_name,
values_to_remove=values_to_remove
    )

    # Step 4: Get the final state
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    get_response = table.get_item(
        Key=key,
        ProjectionExpression=f"#{set_name}",
        ExpressionAttributeNames={f"#{set_name}": set_name},
    )

    # Return all responses
    return {
        "create_response": create_response,
```

```
        "add_response": add_response,
        "remove_response": remove_response,
        "final_state": get_response.get("Item", {}),
    }
```

Example usage of set operations with AWS SDK for Python (Boto3).

```
def example_usage():
    """Example of how to use the set operations functions."""
    # Example parameters
    table_name = "UserPreferences"
    key = {"UserId": "user123"}

    print("Example 1: Creating a string set attribute")
    try:
        response = create_set_attribute(
            table_name=table_name,
            key=key,
            set_name="FavoriteTags",
            set_values=["AWS", "DynamoDB", "NoSQL"],
            set_type="string",
        )
        print(f"Set attribute created successfully: {response.get('Attributes',
 {})}")
    except Exception as e:
        print(f"Error creating set attribute: {e}")

    print("\nExample 2: Adding elements to an existing set")
    try:
        response = add_to_set(
            table_name=table_name,
            key=key,
            set_name="FavoriteTags",
            values_to_add=["Database", "Serverless"],
        )
        print(f"Elements added to set successfully: {response.get('Attributes',
 {})}")
    except Exception as e:
        print(f"Error adding to set: {e}")
```

```
    print("\nExample 3: Removing elements from a set")
    try:
        response = remove_from_set(
            table_name=table_name, key=key, set_name="FavoriteTags",
values_to_remove=["NoSQL"]
        )
        print(f"Elements removed from set successfully:
{response.get('Attributes', {})}")
    except Exception as e:
        print(f"Error removing from set: {e}")

    print("\nExample 4: Demonstrating what happens when you remove the last
element from a set")
    try:
        results = demonstrate_last_element_removal(
            table_name=table_name, key={"UserId": "tempUser"},
set_name="SingleElementSet"
        )

        print(f"Set exists before removal: {results['exists_before']}")
        print(f"Set exists after removal: {results['exists_after']}")

        if not results["exists_after"]:
            print("The set attribute was completely removed when the last element
was deleted.")
        else:
            print("The set attribute still exists after removing the last
element.")
    except Exception as e:
        print(f"Error in last element removal demonstration: {e}")

    print("\nExample 5: Working with a number set")
    try:
        results = work_with_number_set(
            table_name=table_name,
            key={"UserId": "user456"},
            set_name="LuckyNumbers",
            initial_values=[7, 13, 42],
            values_to_add=[99, 100],
            values_to_remove=[13],
        )

        print(f"Initial number set: {results['create_response'].get('Attributes',
{})}")
```

```
        print(f"After adding numbers: {results['add_response'].get('Attributes',
{})}")
        print(f"After removing numbers:
{results['remove_response'].get('Attributes', {})}")
        print(f"Final state: {results['final_state']}")
    except Exception as e:
        print(f"Error working with number set: {e}")

    print("\nKey Points About DynamoDB Sets:")
    print("1. Sets can only contain elements of the same type (string, number, or
binary)")
    print("2. Sets automatically eliminate duplicate values")
    print("3. The ADD operation creates a set if it doesn't exist")
    print("4. The DELETE operation removes specified elements from a set")
    print("5. When the last element is removed from a set, the entire attribute
is deleted")
    print("6. Empty sets are not allowed in DynamoDB")
    print("7. Sets are unordered collections")
    print("8. The ADD operation is atomic for sets")
```

- For API details, see UpdateItem in *AWS SDK for Python (Boto3) API Reference*.

For a complete list of AWS SDK developer guides and code examples, see Using DynamoDB with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Query a DynamoDB table by using batches of PartiQL statements and an AWS SDK

The following code examples show how to:

- Get a batch of items by running multiple SELECT statements.
- Add a batch of items by running multiple INSERT statements.
- Update a batch of items by running multiple UPDATE statements.
- Delete a batch of items by running multiple DELETE statements.

.NET

SDK for .NET

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
// Before you run this example, download 'movies.json' from
// https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/
GettingStarted.Js.02.html,
// and put it in the same folder as the example.

// Separator for the console display.
var SepBar = new string('-', 80);
const string tableName = "movie_table";
const string movieFileName = @"..\..\..\..\..\..\..\..\resources\sample_files
\movies.json";

DisplayInstructions();

// Create the table and wait for it to be active.
Console.WriteLine($"Creating the movie table: {tableName}");

var success = await DynamoDBMethods.CreateMovieTableAsync(tableName);
if (success)
{
    Console.WriteLine($"Successfully created table: {tableName}.");
}

WaitForEnter();

// Add movie information to the table from moviedata.json. See the
// instructions at the top of this file to download the JSON file.
Console.WriteLine($"Inserting movies into the new table. Please wait...");
success = await PartiQLBatchMethods.InsertMovies(tableName, movieFileName);
if (success)
{
    Console.WriteLine("Movies successfully added to the table.");
```

```
}
else
{
    Console.WriteLine("Movies could not be added to the table.");
}

WaitForEnter();

// Update multiple movies by using the BatchExecute statement.
var title1 = "Star Wars";
var year1 = 1977;
var title2 = "Wizard of Oz";
var year2 = 1939;

Console.WriteLine($"Updating two movies with producer information: {title1} and
 {title2}.");
success = await PartiQLBatchMethods.GetBatch(tableName, title1, title2, year1,
 year2);
if (success)
{
    Console.WriteLine($"Successfully retrieved {title1} and {title2}.");
}
else
{
    Console.WriteLine("Select statement failed.");
}

WaitForEnter();

// Update multiple movies by using the BatchExecute statement.
var producer1 = "LucasFilm";
var producer2 = "MGM";

Console.WriteLine($"Updating two movies with producer information: {title1} and
 {title2}.");
success = await PartiQLBatchMethods.UpdateBatch(tableName, producer1, title1,
 year1, producer2, title2, year2);
if (success)
{
    Console.WriteLine($"Successfully updated {title1} and {title2}.");
}
else
{
    Console.WriteLine("Update failed.");
```

```
}

WaitForEnter();

// Delete multiple movies by using the BatchExecute statement.
Console.WriteLine($"Now we will delete {title1} and {title2} from the table.");
success = await PartiQLBatchMethods.DeleteBatch(tableName, title1, year1, title2,
 year2);

if (success)
{
    Console.WriteLine($"Deleted {title1} and {title2}");
}
else
{
    Console.WriteLine($"could not delete {title1} or {title2}");
}

WaitForEnter();

// DNow that the PartiQL Batch scenario is complete, delete the movie table.
success = await DynamoDBMethods.DeleteTableAsync(tableName);

if (success)
{
    Console.WriteLine($"Successfully deleted {tableName}");
}
else
{
    Console.WriteLine($"Could not delete {tableName}");
}

/// <summary>
/// Displays the description of the application on the console.
/// </summary>
void DisplayInstructions()
{
    Console.Clear();
    Console.WriteLine();
    Console.Write(new string(' ', 24));
    Console.WriteLine("DynamoDB PartiQL Basics Example");
    Console.WriteLine(SepBar);
    Console.WriteLine("This demo application shows the basics of using Amazon
 DynamoDB with the AWS SDK for");
```

```
    Console.WriteLine(".NET version 3.7 and .NET 6.");
    Console.WriteLine(SepBar);
    Console.WriteLine("Creates a table by using the CreateTable method.");
    Console.WriteLine("Gets multiple movies by using a PartiQL SELECT
 statement.");
    Console.WriteLine("Updates multiple movies by using the ExecuteBatch
 method.");
    Console.WriteLine("Deletes multiple movies by using a PartiQL DELETE
 statement.");
    Console.WriteLine("Cleans up the resources created for the demo by deleting
 the table.");
    Console.WriteLine(SepBar);

    WaitForEnter();
}

/// <summary>
/// Simple method to wait for the <Enter> key to be pressed.
/// </summary>
void WaitForEnter()
{
    Console.WriteLine("\nPress <Enter> to continue.");
    Console.Write(SepBar);
    _ = Console.ReadLine();
}
```

```
        /// <summary>
        /// Gets movies from the movie table by
        /// using an Amazon DynamoDB PartiQL SELECT statement.
        /// </summary>
        /// <param name="tableName">The name of the table.</param>
        /// <param name="title1">The title of the first movie.</param>
        /// <param name="title2">The title of the second movie.</param>
        /// <param name="year1">The year of the first movie.</param>
        /// <param name="year2">The year of the second movie.</param>
        /// <returns>True if successful.</returns>
        public static async Task<bool> GetBatch(
            string tableName,
            string title1,
            string title2,
            int year1,
            int year2)
        {
```

```
            var getBatch = $"SELECT * FROM {tableName} WHERE title = ? AND year
= ?";
            var statements = new List<BatchStatementRequest>
            {
                new BatchStatementRequest
                {
                    Statement = getBatch,
                    Parameters = new List<AttributeValue>
                    {
                        new AttributeValue { S = title1 },
                        new AttributeValue { N = year1.ToString() },
                    },
                },

                new BatchStatementRequest
                {
                    Statement = getBatch,
                    Parameters = new List<AttributeValue>
                    {
                        new AttributeValue { S = title2 },
                        new AttributeValue { N = year2.ToString() },
                    },
                }
            };

            var response = await Client.BatchExecuteStatementAsync(new
BatchExecuteStatementRequest
            {
                Statements = statements,
            });

            if (response.Responses.Count > 0)
            {
                response.Responses.ForEach(r =>
                {
                    if (r.Item.Any())
                    {

Console.WriteLine($"{r.Item["title"]}\t{r.Item["year"]}");
                    }
                });
                return true;
            }
            else
```

```
            {
                Console.WriteLine($"Couldn't find either {title1} or {title2}.");
                return false;
            }

        }

        /// <summary>
        /// Inserts movies imported from a JSON file into the movie table by
        /// using an Amazon DynamoDB PartiQL INSERT statement.
        /// </summary>
        /// <param name="tableName">The name of the table into which the movie
        /// information will be inserted.</param>
        /// <param name="movieFileName">The name of the JSON file that contains
        /// movie information.</param>
        /// <returns>A Boolean value that indicates the success or failure of
        /// the insert operation.</returns>
        public static async Task<bool> InsertMovies(string tableName, string
movieFileName)
        {
            // Get the list of movies from the JSON file.
            var movies = ImportMovies(movieFileName);

            var success = false;

            if (movies is not null)
            {
                // Insert the movies in a batch using PartiQL. Because the
                // batch can contain a maximum of 25 items, insert 25 movies
                // at a time.
                string insertBatch = $"INSERT INTO {tableName} VALUE
{{'title': ?, 'year': ?}}";
                var statements = new List<BatchStatementRequest>();

                try
                {
                    for (var indexOffset = 0; indexOffset < 250; indexOffset +=
25)
                    {
                        for (var i = indexOffset; i < indexOffset + 25; i++)
                        {
                            statements.Add(new BatchStatementRequest
                            {
                                Statement = insertBatch,
```

```
                                Parameters = new List<AttributeValue>
                                {
                                    new AttributeValue { S = movies[i].Title },
                                    new AttributeValue { N =
movies[i].Year.ToString() },
                                },
                            });
                        }

                        var response = await
Client.BatchExecuteStatementAsync(new BatchExecuteStatementRequest
                        {
                            Statements = statements,
                        });

                        // Wait between batches for movies to be successfully
added.
                        System.Threading.Thread.Sleep(3000);

                        success = response.HttpStatusCode ==
System.Net.HttpStatusCode.OK;

                        // Clear the list of statements for the next batch.
                        statements.Clear();
                    }
                }
                catch (AmazonDynamoDBException ex)
                {
                    Console.WriteLine(ex.Message);
                }
            }

            return success;
        }

        /// <summary>
        /// Loads the contents of a JSON file into a list of movies to be
        /// added to the DynamoDB table.
        /// </summary>
        /// <param name="movieFileName">The full path to the JSON file.</param>
        /// <returns>A generic list of movie objects.</returns>
        public static List<Movie> ImportMovies(string movieFileName)
        {
            if (!File.Exists(movieFileName))
```

```
            {
                return null!;
            }

            using var sr = new StreamReader(movieFileName);
            string json = sr.ReadToEnd();
            var allMovies = JsonConvert.DeserializeObject<List<Movie>>(json);

            if (allMovies is not null)
            {
                // Return the first 250 entries.
                return allMovies.GetRange(0, 250);
            }
            else
            {
                return null!;
            }
        }

        /// <summary>
        /// Updates information for multiple movies.
        /// </summary>
        /// <param name="tableName">The name of the table containing the
        /// movies to be updated.</param>
        /// <param name="producer1">The producer name for the first movie
        /// to update.</param>
        /// <param name="title1">The title of the first movie.</param>
        /// <param name="year1">The year that the first movie was released.</
param>
        /// <param name="producer2">The producer name for the second
        /// movie to update.</param>
        /// <param name="title2">The title of the second movie.</param>
        /// <param name="year2">The year that the second movie was released.</
param>
        /// <returns>A Boolean value that indicates the success of the update.</
returns>
        public static async Task<bool> UpdateBatch(
            string tableName,
            string producer1,
            string title1,
            int year1,
            string producer2,
            string title2,
            int year2)
```

```
        {

            string updateBatch = $"UPDATE {tableName} SET Producer=? WHERE title
= ? AND year = ?";
            var statements = new List<BatchStatementRequest>
            {
                new BatchStatementRequest
                {
                    Statement = updateBatch,
                    Parameters = new List<AttributeValue>
                    {
                        new AttributeValue { S = producer1 },
                        new AttributeValue { S = title1 },
                        new AttributeValue { N = year1.ToString() },
                    },
                },

                new BatchStatementRequest
                {
                    Statement = updateBatch,
                    Parameters = new List<AttributeValue>
                    {
                        new AttributeValue { S = producer2 },
                        new AttributeValue { S = title2 },
                        new AttributeValue { N = year2.ToString() },
                    },
                }
            };

            var response = await Client.BatchExecuteStatementAsync(new
BatchExecuteStatementRequest
            {
                Statements = statements,
            });

            return response.HttpStatusCode == System.Net.HttpStatusCode.OK;
        }

        /// <summary>
        /// Deletes multiple movies using a PartiQL BatchExecuteAsync
        /// statement.
        /// </summary>
        /// <param name="tableName">The name of the table containing the
        /// moves that will be deleted.</param>
```

```
        /// <param name="title1">The title of the first movie.</param>
        /// <param name="year1">The year the first movie was released.</param>
        /// <param name="title2">The title of the second movie.</param>
        /// <param name="year2">The year the second movie was released.</param>
        /// <returns>A Boolean value indicating the success of the operation.</
returns>
        public static async Task<bool> DeleteBatch(
            string tableName,
            string title1,
            int year1,
            string title2,
            int year2)
        {

            string updateBatch = $"DELETE FROM {tableName} WHERE title = ? AND
 year = ?";
            var statements = new List<BatchStatementRequest>
            {
                new BatchStatementRequest
                {
                    Statement = updateBatch,
                    Parameters = new List<AttributeValue>
                    {
                        new AttributeValue { S = title1 },
                        new AttributeValue { N = year1.ToString() },
                    },
                },

                new BatchStatementRequest
                {
                    Statement = updateBatch,
                    Parameters = new List<AttributeValue>
                    {
                        new AttributeValue { S = title2 },
                        new AttributeValue { N = year2.ToString() },
                    },
                }
            };

            var response = await Client.BatchExecuteStatementAsync(new
 BatchExecuteStatementRequest
            {
                Statements = statements,
            });
```

```
            return response.HttpStatusCode == System.Net.HttpStatusCode.OK;
        }
```

- For API details, see BatchExecuteStatement in *AWS SDK for .NET API Reference*.

C++

SDK for C++

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
        Aws::Client::ClientConfiguration clientConfig;
        //  1. Create a table. (CreateTable)
        if (AwsDoc::DynamoDB::createMoviesDynamoDBTable(clientConfig)) {

            AwsDoc::DynamoDB::partiqlBatchExecuteScenario(clientConfig);

            // 7. Delete the table. (DeleteTable)
            AwsDoc::DynamoDB::deleteMoviesDynamoDBTable(clientConfig);
        }

//! Scenario to modify and query a DynamoDB table using PartiQL batch statements.
/*!
  \sa partiqlBatchExecuteScenario()
  \param clientConfiguration: AWS client configuration.
  \return bool: Function succeeded.
 */
bool AwsDoc::DynamoDB::partiqlBatchExecuteScenario(
        const Aws::Client::ClientConfiguration &clientConfiguration) {

    // 2. Add multiple movies using "Insert" statements. (BatchExecuteStatement)
    Aws::DynamoDB::DynamoDBClient dynamoClient(clientConfiguration);

    std::vector<Aws::String> titles;
    std::vector<float> ratings;
    std::vector<int> years;
```

```
    std::vector<Aws::String> plots;
    Aws::String doAgain = "n";
    do {
        Aws::String aTitle = askQuestion(
                "Enter the title of a movie you want to add to the table: ");
        titles.push_back(aTitle);
        int aYear = askQuestionForInt("What year was it released? ");
        years.push_back(aYear);
        float aRating = askQuestionForFloatRange(
                "On a scale of 1 - 10, how do you rate it? ",
                1, 10);
        ratings.push_back(aRating);
        Aws::String aPlot = askQuestion("Summarize the plot for me: ");
        plots.push_back(aPlot);

        doAgain = askQuestion(Aws::String("Would you like to add more movies? (y/
n) "));
    } while (doAgain == "y");

    std::cout << "Adding " << titles.size()
              << (titles.size() == 1 ? " movie " : " movies ")
              << "to the table using a batch \"INSERT\" statement." << std::endl;

    {
        Aws::Vector<Aws::DynamoDB::Model::BatchStatementRequest> statements(
                titles.size());

        std::stringstream sqlStream;
        sqlStream << "INSERT INTO \"" << MOVIE_TABLE_NAME << "\" VALUE {'"
                  << TITLE_KEY << "': ?, '" << YEAR_KEY << "': ?, '"
                  << INFO_KEY << "': ?}";

        std::string sql(sqlStream.str());

        for (size_t i = 0; i < statements.size(); ++i) {
            statements[i].SetStatement(sql);

            Aws::Vector<Aws::DynamoDB::Model::AttributeValue> attributes;
            attributes.push_back(
                    Aws::DynamoDB::Model::AttributeValue().SetS(titles[i]));

 attributes.push_back(Aws::DynamoDB::Model::AttributeValue().SetN(years[i]));

            // Create attribute for the info map.
```

```
            Aws::DynamoDB::Model::AttributeValue infoMapAttribute;

            std::shared_ptr<Aws::DynamoDB::Model::AttributeValue> ratingAttribute
= Aws::MakeShared<Aws::DynamoDB::Model::AttributeValue>(
                    ALLOCATION_TAG.c_str());
            ratingAttribute->SetN(ratings[i]);
            infoMapAttribute.AddMEntry(RATING_KEY, ratingAttribute);

            std::shared_ptr<Aws::DynamoDB::Model::AttributeValue> plotAttribute =
Aws::MakeShared<Aws::DynamoDB::Model::AttributeValue>(
                    ALLOCATION_TAG.c_str());
            plotAttribute->SetS(plots[i]);
            infoMapAttribute.AddMEntry(PLOT_KEY, plotAttribute);
            attributes.push_back(infoMapAttribute);
            statements[i].SetParameters(attributes);
        }

        Aws::DynamoDB::Model::BatchExecuteStatementRequest request;

        request.SetStatements(statements);

        Aws::DynamoDB::Model::BatchExecuteStatementOutcome outcome =
dynamoClient.BatchExecuteStatement(
                request);
        if (!outcome.IsSuccess()) {
            std::cerr << "Failed to add the movies: " <<
outcome.GetError().GetMessage()
                      << std::endl;
            return false;
        }
    }

    std::cout << "Retrieving the movie data with a batch \"SELECT\" statement."
              << std::endl;

    // 3. Get the data for multiple movies using "Select" statements.
(BatchExecuteStatement)
    {
        Aws::Vector<Aws::DynamoDB::Model::BatchStatementRequest> statements(
                titles.size());
        std::stringstream sqlStream;
        sqlStream << "SELECT * FROM  \"" << MOVIE_TABLE_NAME << "\" WHERE "
                  << TITLE_KEY << "=? and " << YEAR_KEY << "=?";
```

```
        std::string sql(sqlStream.str());

        for (size_t i = 0; i < statements.size(); ++i) {
            statements[i].SetStatement(sql);
            Aws::Vector<Aws::DynamoDB::Model::AttributeValue> attributes;
            attributes.push_back(
                    Aws::DynamoDB::Model::AttributeValue().SetS(titles[i]));

attributes.push_back(Aws::DynamoDB::Model::AttributeValue().SetN(years[i]));
            statements[i].SetParameters(attributes);
        }

        Aws::DynamoDB::Model::BatchExecuteStatementRequest request;

        request.SetStatements(statements);

        Aws::DynamoDB::Model::BatchExecuteStatementOutcome outcome =
dynamoClient.BatchExecuteStatement(
                request);
        if (outcome.IsSuccess()) {
            const Aws::DynamoDB::Model::BatchExecuteStatementResult &result =
outcome.GetResult();

            const Aws::Vector<Aws::DynamoDB::Model::BatchStatementResponse>
&responses = result.GetResponses();

            for (const Aws::DynamoDB::Model::BatchStatementResponse &response:
responses) {
                const Aws::Map<Aws::String, Aws::DynamoDB::Model::AttributeValue>
&item = response.GetItem();

                printMovieInfo(item);
            }
        }
        else {
            std::cerr << "Failed to retrieve the movie information: "
                      << outcome.GetError().GetMessage() << std::endl;
            return false;
        }
    }

    // 4. Update the data for multiple movies using "Update" statements.
(BatchExecuteStatement)
```

```
   for (size_t i = 0; i < titles.size(); ++i) {
       ratings[i] = askQuestionForFloatRange(
               Aws::String("\nLet's update your the movie, \"") + titles[i] +
               ".\nYou rated it  " + std::to_string(ratings[i])
               + ", what new rating would you give it? ", 1, 10);
   }

   std::cout << "Updating the movie with a batch \"UPDATE\" statement." <<
std::endl;

   {
       Aws::Vector<Aws::DynamoDB::Model::BatchStatementRequest> statements(
               titles.size());

       std::stringstream sqlStream;
       sqlStream << "UPDATE \"" << MOVIE_TABLE_NAME << "\" SET "
                 << INFO_KEY << "." << RATING_KEY << "=? WHERE "
                 << TITLE_KEY << "=? AND " << YEAR_KEY << "=?";


       std::string sql(sqlStream.str());

       for (size_t i = 0; i < statements.size(); ++i) {
           statements[i].SetStatement(sql);

           Aws::Vector<Aws::DynamoDB::Model::AttributeValue> attributes;
           attributes.push_back(
                   Aws::DynamoDB::Model::AttributeValue().SetN(ratings[i]));
           attributes.push_back(
                   Aws::DynamoDB::Model::AttributeValue().SetS(titles[i]));

attributes.push_back(Aws::DynamoDB::Model::AttributeValue().SetN(years[i]));
           statements[i].SetParameters(attributes);
       }

       Aws::DynamoDB::Model::BatchExecuteStatementRequest request;

       request.SetStatements(statements);
       Aws::DynamoDB::Model::BatchExecuteStatementOutcome outcome =
dynamoClient.BatchExecuteStatement(
               request);
       if (!outcome.IsSuccess()) {
           std::cerr << "Failed to update movie information: "
                     << outcome.GetError().GetMessage() << std::endl;
```

```
            return false;
        }
    }

    std::cout << "Retrieving the updated movie data with a batch \"SELECT\"
statement."
              << std::endl;

    // 5. Get the updated data for multiple movies using "Select" statements.
(BatchExecuteStatement)
    {
        Aws::Vector<Aws::DynamoDB::Model::BatchStatementRequest> statements(
                titles.size());
        std::stringstream sqlStream;
        sqlStream << "SELECT * FROM  \"" << MOVIE_TABLE_NAME << "\" WHERE "
                  << TITLE_KEY << "=? and " << YEAR_KEY << "=?";

        std::string sql(sqlStream.str());

        for (size_t i = 0; i < statements.size(); ++i) {
            statements[i].SetStatement(sql);
            Aws::Vector<Aws::DynamoDB::Model::AttributeValue> attributes;
            attributes.push_back(
                    Aws::DynamoDB::Model::AttributeValue().SetS(titles[i]));

attributes.push_back(Aws::DynamoDB::Model::AttributeValue().SetN(years[i]));
            statements[i].SetParameters(attributes);
        }

        Aws::DynamoDB::Model::BatchExecuteStatementRequest request;

        request.SetStatements(statements);

        Aws::DynamoDB::Model::BatchExecuteStatementOutcome outcome =
dynamoClient.BatchExecuteStatement(
                request);
        if (outcome.IsSuccess()) {
            const Aws::DynamoDB::Model::BatchExecuteStatementResult &result =
outcome.GetResult();

            const Aws::Vector<Aws::DynamoDB::Model::BatchStatementResponse>
&responses = result.GetResponses();
```

```
            for (const Aws::DynamoDB::Model::BatchStatementResponse &response:
responses) {
                const Aws::Map<Aws::String, Aws::DynamoDB::Model::AttributeValue>
&item = response.GetItem();

                printMovieInfo(item);
            }
        }
        else {
            std::cerr << "Failed to retrieve the movies information: "
                      << outcome.GetError().GetMessage() << std::endl;
            return false;
        }
    }

    std::cout << "Deleting the movie data with a batch \"DELETE\" statement."
              << std::endl;

    // 6. Delete multiple movies using "Delete" statements.
(BatchExecuteStatement)
    {
        Aws::Vector<Aws::DynamoDB::Model::BatchStatementRequest> statements(
                titles.size());
        std::stringstream sqlStream;
        sqlStream << "DELETE FROM  \"" << MOVIE_TABLE_NAME << "\" WHERE "
                  << TITLE_KEY << "=? and " << YEAR_KEY << "=?";

        std::string sql(sqlStream.str());

        for (size_t i = 0; i < statements.size(); ++i) {
            statements[i].SetStatement(sql);
            Aws::Vector<Aws::DynamoDB::Model::AttributeValue> attributes;
            attributes.push_back(
                    Aws::DynamoDB::Model::AttributeValue().SetS(titles[i]));

attributes.push_back(Aws::DynamoDB::Model::AttributeValue().SetN(years[i]));
            statements[i].SetParameters(attributes);
        }

        Aws::DynamoDB::Model::BatchExecuteStatementRequest request;

        request.SetStatements(statements);
```

```
        Aws::DynamoDB::Model::BatchExecuteStatementOutcome outcome =
 dynamoClient.BatchExecuteStatement(
                request);

        if (!outcome.IsSuccess()) {
            std::cerr << "Failed to delete the movies: "
                      << outcome.GetError().GetMessage() << std::endl;
            return false;
        }
    }

    return true;
}

//! Create a DynamoDB table to be used in sample code scenarios.
/*!
  \sa createMoviesDynamoDBTable()
  \param clientConfiguration: AWS client configuration.
  \return bool: Function succeeded.
*/
bool AwsDoc::DynamoDB::createMoviesDynamoDBTable(
        const Aws::Client::ClientConfiguration &clientConfiguration) {
    Aws::DynamoDB::DynamoDBClient dynamoClient(clientConfiguration);

    bool movieTableAlreadyExisted = false;

    {
        Aws::DynamoDB::Model::CreateTableRequest request;

        Aws::DynamoDB::Model::AttributeDefinition yearAttributeDefinition;
        yearAttributeDefinition.SetAttributeName(YEAR_KEY);
        yearAttributeDefinition.SetAttributeType(
                Aws::DynamoDB::Model::ScalarAttributeType::N);
        request.AddAttributeDefinitions(yearAttributeDefinition);

        Aws::DynamoDB::Model::AttributeDefinition titleAttributeDefinition;
        yearAttributeDefinition.SetAttributeName(TITLE_KEY);
        yearAttributeDefinition.SetAttributeType(
                Aws::DynamoDB::Model::ScalarAttributeType::S);
        request.AddAttributeDefinitions(yearAttributeDefinition);

        Aws::DynamoDB::Model::KeySchemaElement yearKeySchema;
        yearKeySchema.WithAttributeName(YEAR_KEY).WithKeyType(
                Aws::DynamoDB::Model::KeyType::HASH);
```

```
        request.AddKeySchema(yearKeySchema);

        Aws::DynamoDB::Model::KeySchemaElement titleKeySchema;
        yearKeySchema.WithAttributeName(TITLE_KEY).WithKeyType(
                Aws::DynamoDB::Model::KeyType::RANGE);
        request.AddKeySchema(yearKeySchema);

        Aws::DynamoDB::Model::ProvisionedThroughput throughput;
        throughput.WithReadCapacityUnits(
                PROVISIONED_THROUGHPUT_UNITS).WithWriteCapacityUnits(
                PROVISIONED_THROUGHPUT_UNITS);
        request.SetProvisionedThroughput(throughput);
        request.SetTableName(MOVIE_TABLE_NAME);

        std::cout << "Creating table '" << MOVIE_TABLE_NAME << "'..." <<
std::endl;
        const Aws::DynamoDB::Model::CreateTableOutcome &result =
dynamoClient.CreateTable(
                request);
        if (!result.IsSuccess()) {
            if (result.GetError().GetErrorType() ==
                Aws::DynamoDB::DynamoDBErrors::RESOURCE_IN_USE) {
                std::cout << "Table already exists." << std::endl;
                movieTableAlreadyExisted = true;
            }
            else {
                std::cerr << "Failed to create table: "
                          << result.GetError().GetMessage();
                return false;
            }
        }
    }

    // Wait for table to become active.
    if (!movieTableAlreadyExisted) {
        std::cout << "Waiting for table '" << MOVIE_TABLE_NAME
                  << "' to become active...." << std::endl;
        if (!AwsDoc::DynamoDB::waitTableActive(MOVIE_TABLE_NAME,
clientConfiguration)) {
            return false;
        }
        std::cout << "Table '" << MOVIE_TABLE_NAME << "' created and active."
                  << std::endl;
    }
```

```
    return true;
}

//! Delete the DynamoDB table used for sample code scenarios.
/*!
  \sa deleteMoviesDynamoDBTable()
  \param clientConfiguration: AWS client configuration.
  \return bool: Function succeeded.
*/
bool AwsDoc::DynamoDB::deleteMoviesDynamoDBTable(
        const Aws::Client::ClientConfiguration &clientConfiguration) {
    Aws::DynamoDB::DynamoDBClient dynamoClient(clientConfiguration);

    Aws::DynamoDB::Model::DeleteTableRequest request;
    request.SetTableName(MOVIE_TABLE_NAME);

    const Aws::DynamoDB::Model::DeleteTableOutcome &result =
 dynamoClient.DeleteTable(
            request);
    if (result.IsSuccess()) {
        std::cout << "Your table \""
                  << result.GetResult().GetTableDescription().GetTableName()
                  << " was deleted.\n";
    }
    else {
        std::cerr << "Failed to delete table: " << result.GetError().GetMessage()
                  << std::endl;
    }

    return result.IsSuccess();
}

//! Query a newly created DynamoDB table until it is active.
/*!
  \sa waitTableActive()
  \param waitTableActive: The DynamoDB table's name.
  \param dynamoClient: A DynamoDB client.
  \return bool: Function succeeded.
*/
bool AwsDoc::DynamoDB::waitTableActive(const Aws::String &tableName,
                                       const Aws::DynamoDB::DynamoDBClient
 &dynamoClient) {
```

```
    // Repeatedly call DescribeTable until table is ACTIVE.
    const int MAX_QUERIES = 20;
    Aws::DynamoDB::Model::DescribeTableRequest request;
    request.SetTableName(tableName);

    int count = 0;
    while (count < MAX_QUERIES) {
        const Aws::DynamoDB::Model::DescribeTableOutcome &result =
 dynamoClient.DescribeTable(
                request);
        if (result.IsSuccess()) {
            Aws::DynamoDB::Model::TableStatus status =
 result.GetResult().GetTable().GetTableStatus();

            if (Aws::DynamoDB::Model::TableStatus::ACTIVE != status) {
                std::this_thread::sleep_for(std::chrono::seconds(1));
            }
            else {
                return true;
            }
        }
        else {
            std::cerr << "Error DynamoDB::waitTableActive "
                      << result.GetError().GetMessage() << std::endl;
            return false;
        }
        count++;
    }
    return false;
}
```

- For API details, see BatchExecuteStatement in *AWS SDK for C++ API Reference*.

Go

SDK for Go V2

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Run a scenario that creates a table and runs batches of PartiQL queries.

```
import (
 "context"
 "fmt"
 "log"
 "strings"
 "time"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb"
 "github.com/awsdocs/aws-doc-sdk-examples/gov2/dynamodb/actions"
)

// RunPartiQLBatchScenario shows you how to use the AWS SDK for Go
// to run batches of PartiQL statements to query a table that stores data about
 movies.
//
//   - Use batches of PartiQL statements to add, get, update, and delete data for
//     individual movies.
//
// This example creates an Amazon DynamoDB service client from the specified
 sdkConfig so that
// you can replace it with a mocked or stubbed config for unit testing.
//
// This example creates and deletes a DynamoDB table to use during the scenario.
func RunPartiQLBatchScenario(ctx context.Context, sdkConfig aws.Config, tableName
 string) {
 defer func() {
  if r := recover(); r != nil {
   fmt.Printf("Something went wrong with the demo.")
  }
 }()

 log.Println(strings.Repeat("-", 88))
 log.Println("Welcome to the Amazon DynamoDB PartiQL batch demo.")
 log.Println(strings.Repeat("-", 88))

 tableBasics := actions.TableBasics{
  DynamoDbClient: dynamodb.NewFromConfig(sdkConfig),
  TableName:      tableName,
 }
 runner := actions.PartiQLRunner{
```

```
 DynamoDbClient: dynamodb.NewFromConfig(sdkConfig),
 TableName:      tableName,
}

exists, err := tableBasics.TableExists(ctx)
if err != nil {
 panic(err)
}
if !exists {
 log.Printf("Creating table %v...\n", tableName)
 _, err = tableBasics.CreateMovieTable(ctx)
 if err != nil {
  panic(err)
 } else {
  log.Printf("Created table %v.\n", tableName)
 }
} else {
 log.Printf("Table %v already exists.\n", tableName)
}
log.Println(strings.Repeat("-", 88))

currentYear, _, _ := time.Now().Date()
customMovies := []actions.Movie{{
 Title: "House PartiQL",
 Year:  currentYear - 5,
 Info: map[string]interface{}{
  "plot":   "Wacky high jinks result from querying a mysterious database.",
  "rating": 8.5}}, {
 Title: "House PartiQL 2",
 Year:  currentYear - 3,
 Info: map[string]interface{}{
  "plot":   "Moderate high jinks result from querying another mysterious
database.",
  "rating": 6.5}}, {
 Title: "House PartiQL 3",
 Year:  currentYear - 1,
 Info: map[string]interface{}{
  "plot":   "Tepid high jinks result from querying yet another mysterious
database.",
  "rating": 2.5},
},
}

log.Printf("Inserting a batch of movies into table '%v'.\n", tableName)
```

```
err = runner.AddMovieBatch(ctx, customMovies)
if err == nil {
 log.Printf("Added %v movies to the table.\n", len(customMovies))
}
log.Println(strings.Repeat("-", 88))

log.Println("Getting data for a batch of movies.")
movies, err := runner.GetMovieBatch(ctx, customMovies)
if err == nil {
 for _, movie := range movies {
  log.Println(movie)
 }
}
log.Println(strings.Repeat("-", 88))

newRatings := []float64{7.7, 4.4, 1.1}
log.Println("Updating a batch of movies with new ratings.")
err = runner.UpdateMovieBatch(ctx, customMovies, newRatings)
if err == nil {
 log.Printf("Updated %v movies with new ratings.\n", len(customMovies))
}
log.Println(strings.Repeat("-", 88))

log.Println("Getting projected data from the table to verify our update.")
log.Println("Using a page size of 2 to demonstrate paging.")
projections, err := runner.GetAllMovies(ctx, 2)
if err == nil {
 log.Println("All movies:")
 for _, projection := range projections {
  log.Println(projection)
 }
}
log.Println(strings.Repeat("-", 88))

log.Println("Deleting a batch of movies.")
err = runner.DeleteMovieBatch(ctx, customMovies)
if err == nil {
 log.Printf("Deleted %v movies.\n", len(customMovies))
}

err = tableBasics.DeleteTable(ctx)
if err == nil {
 log.Printf("Deleted table %v.\n", tableBasics.TableName)
}
```

```
 log.Println(strings.Repeat("-", 88))
 log.Println("Thanks for watching!")
 log.Println(strings.Repeat("-", 88))
}
```

Define a Movie struct that is used in this example.

```
import (
 "archive/zip"
 "bytes"
 "encoding/json"
 "fmt"
 "io"
 "log"
 "net/http"

 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/attributevalue"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb/types"
)

// Movie encapsulates data about a movie. Title and Year are the composite
 primary key
// of the movie in Amazon DynamoDB. Title is the sort key, Year is the partition
 key,
// and Info is additional data.
type Movie struct {
 Title string                `dynamodbav:"title"`
 Year  int                   `dynamodbav:"year"`
 Info  map[string]interface{} `dynamodbav:"info"`
}

// GetKey returns the composite primary key of the movie in a format that can be
// sent to DynamoDB.
func (movie Movie) GetKey() map[string]types.AttributeValue {
 title, err := attributevalue.Marshal(movie.Title)
 if err != nil {
  panic(err)
 }
 year, err := attributevalue.Marshal(movie.Year)
```

```
 if err != nil {
  panic(err)
 }
 return map[string]types.AttributeValue{"title": title, "year": year}
}

// String returns the title, year, rating, and plot of a movie, formatted for the
 example.
func (movie Movie) String() string {
 return fmt.Sprintf("%v\n\tReleased: %v\n\tRating: %v\n\tPlot: %v\n",
  movie.Title, movie.Year, movie.Info["rating"], movie.Info["plot"])
}
```

Create a struct and methods that run PartiQL statements.

```
import (
 "context"
 "fmt"
 "log"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/attributevalue"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb/types"
)

// PartiQLRunner encapsulates the Amazon DynamoDB service actions used in the
// PartiQL examples. It contains a DynamoDB service client that is used to act on
 the
// specified table.
type PartiQLRunner struct {
 DynamoDbClient *dynamodb.Client
 TableName      string
}



// AddMovieBatch runs a batch of PartiQL INSERT statements to add multiple movies
 to the
// DynamoDB table.
```

```
func (runner PartiQLRunner) AddMovieBatch(ctx context.Context, movies []Movie)
 error {
 statementRequests := make([]types.BatchStatementRequest, len(movies))
 for index, movie := range movies {
  params, err := attributevalue.MarshalList([]interface{}{movie.Title,
 movie.Year, movie.Info})
  if err != nil {
   panic(err)
  }
  statementRequests[index] = types.BatchStatementRequest{
   Statement: aws.String(fmt.Sprintf(
    "INSERT INTO \"%v\" VALUE {'title': ?, 'year': ?, 'info': ?}",
 runner.TableName)),
   Parameters: params,
  }
 }

 _, err := runner.DynamoDbClient.BatchExecuteStatement(ctx,
 &dynamodb.BatchExecuteStatementInput{
  Statements: statementRequests,
 })
 if err != nil {
  log.Printf("Couldn't insert a batch of items with PartiQL. Here's why: %v\n",
 err)
 }
 return err
}



// GetMovieBatch runs a batch of PartiQL SELECT statements to get multiple movies
 from
// the DynamoDB table by title and year.
func (runner PartiQLRunner) GetMovieBatch(ctx context.Context, movies []Movie)
 ([]Movie, error) {
 statementRequests := make([]types.BatchStatementRequest, len(movies))
 for index, movie := range movies {
  params, err := attributevalue.MarshalList([]interface{}{movie.Title,
 movie.Year})
  if err != nil {
   panic(err)
  }
  statementRequests[index] = types.BatchStatementRequest{
   Statement: aws.String(
```

```
    fmt.Sprintf("SELECT * FROM \"%v\" WHERE title=? AND year=?",
 runner.TableName)),
   Parameters: params,
  }
 }

 output, err := runner.DynamoDbClient.BatchExecuteStatement(ctx,
 &dynamodb.BatchExecuteStatementInput{
  Statements: statementRequests,
 })
 var outMovies []Movie
 if err != nil {
  log.Printf("Couldn't get a batch of items with PartiQL. Here's why: %v\n", err)
 } else {
  for _, response := range output.Responses {
   var movie Movie
   err = attributevalue.UnmarshalMap(response.Item, &movie)
   if err != nil {
    log.Printf("Couldn't unmarshal response. Here's why: %v\n", err)
   } else {
    outMovies = append(outMovies, movie)
   }
  }
 }
 return outMovies, err
}



// GetAllMovies runs a PartiQL SELECT statement to get all movies from the
 DynamoDB table.
// pageSize is not typically required and is used to show how to paginate the
 results.
// The results are projected to return only the title and rating of each movie.
func (runner PartiQLRunner) GetAllMovies(ctx context.Context, pageSize int32)
 ([]map[string]interface{}, error) {
 var output []map[string]interface{}
 var response *dynamodb.ExecuteStatementOutput
 var err error
 var nextToken *string
 for moreData := true; moreData; {
  response, err = runner.DynamoDbClient.ExecuteStatement(ctx,
 &dynamodb.ExecuteStatementInput{
   Statement: aws.String(
```

```
    fmt.Sprintf("SELECT title, info.rating FROM \"%v\"", runner.TableName)),
   Limit:     aws.Int32(pageSize),
   NextToken: nextToken,
  })
  if err != nil {
   log.Printf("Couldn't get movies. Here's why: %v\n", err)
   moreData = false
  } else {
   var pageOutput []map[string]interface{}
   err = attributevalue.UnmarshalListOfMaps(response.Items, &pageOutput)
   if err != nil {
    log.Printf("Couldn't unmarshal response. Here's why: %v\n", err)
   } else {
    log.Printf("Got a page of length %v.\n", len(response.Items))
    output = append(output, pageOutput...)
   }
   nextToken = response.NextToken
   moreData = nextToken != nil
  }
 }
 return output, err
}



// UpdateMovieBatch runs a batch of PartiQL UPDATE statements to update the
 rating of
// multiple movies that already exist in the DynamoDB table.
func (runner PartiQLRunner) UpdateMovieBatch(ctx context.Context, movies []Movie,
 ratings []float64) error {
 statementRequests := make([]types.BatchStatementRequest, len(movies))
 for index, movie := range movies {
  params, err := attributevalue.MarshalList([]interface{}{ratings[index],
 movie.Title, movie.Year})
  if err != nil {
   panic(err)
  }
  statementRequests[index] = types.BatchStatementRequest{
   Statement: aws.String(
    fmt.Sprintf("UPDATE \"%v\" SET info.rating=? WHERE title=? AND year=?",
 runner.TableName)),
   Parameters: params,
  }
 }
```

```
 _, err := runner.DynamoDbClient.BatchExecuteStatement(ctx,
 &dynamodb.BatchExecuteStatementInput{
  Statements: statementRequests,
 })
 if err != nil {
  log.Printf("Couldn't update the batch of movies. Here's why: %v\n", err)
 }
 return err
}



// DeleteMovieBatch runs a batch of PartiQL DELETE statements to remove multiple
 movies
// from the DynamoDB table.
func (runner PartiQLRunner) DeleteMovieBatch(ctx context.Context, movies []Movie)
 error {
 statementRequests := make([]types.BatchStatementRequest, len(movies))
 for index, movie := range movies {
  params, err := attributevalue.MarshalList([]interface{}{movie.Title,
 movie.Year})
  if err != nil {
   panic(err)
  }
  statementRequests[index] = types.BatchStatementRequest{
   Statement: aws.String(
    fmt.Sprintf("DELETE FROM \"%v\" WHERE title=? AND year=?",
 runner.TableName)),
   Parameters: params,
  }
 }

 _, err := runner.DynamoDbClient.BatchExecuteStatement(ctx,
 &dynamodb.BatchExecuteStatementInput{
  Statements: statementRequests,
 })
 if err != nil {
  log.Printf("Couldn't delete the batch of movies. Here's why: %v\n", err)
 }
 return err
}
```

- For API details, see BatchExecuteStatement in *AWS SDK for Go API Reference*.

Java

SDK for Java 2.x

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
public class ScenarioPartiQLBatch {
    public static void main(String[] args) throws IOException {
        String tableName = "MoviesPartiQBatch";
        Region region = Region.US_EAST_1;
        DynamoDbClient ddb = DynamoDbClient.builder()
            .region(region)
            .build();

        System.out.println("Creating an Amazon DynamoDB table named " + tableName
            + " with a key named year and a sort key named title.");
        createTable(ddb, tableName);

        System.out.println("Adding multiple records into the " + tableName
            + " table using a batch command.");
        putRecordBatch(ddb);

        // Update multiple movies by using the BatchExecute statement.
        String title1 = "Star Wars";
        int year1 = 1977;
        String title2 = "Wizard of Oz";
        int year2 = 1939;

        System.out.println("Query two movies.");
        getBatch(ddb, tableName, title1, title2, year1, year2);

        System.out.println("Updating multiple records using a batch command.");
        updateTableItemBatch(ddb);
```

```
        System.out.println("Deleting multiple records using a batch command.");
        deleteItemBatch(ddb);

        System.out.println("Deleting the Amazon DynamoDB table.");
        deleteDynamoDBTable(ddb, tableName);
        ddb.close();
    }

    public static boolean getBatch(DynamoDbClient ddb, String tableName, String
title1, String title2, int year1, int year2) {
        String getBatch = "SELECT * FROM " + tableName + " WHERE title = ? AND
year = ?";

        List<BatchStatementRequest> statements = new ArrayList<>();
        statements.add(BatchStatementRequest.builder()
            .statement(getBatch)
            .parameters(AttributeValue.builder().s(title1).build(),
                AttributeValue.builder().n(String.valueOf(year1)).build())
            .build());
        statements.add(BatchStatementRequest.builder()
            .statement(getBatch)
            .parameters(AttributeValue.builder().s(title2).build(),
                AttributeValue.builder().n(String.valueOf(year2)).build())
            .build());

        BatchExecuteStatementRequest batchExecuteStatementRequest =
BatchExecuteStatementRequest.builder()
            .statements(statements)
            .build();

        try {
            BatchExecuteStatementResponse response =
ddb.batchExecuteStatement(batchExecuteStatementRequest);
            if (!response.responses().isEmpty()) {
                response.responses().forEach(r -> {
                    System.out.println(r.item().get("title") + "\\t" +
r.item().get("year"));
                });
                return true;
            } else {
                System.out.println("Couldn't find either " + title1 + " or " +
title2 + ".");
                return false;
            }
```

```
        } catch (DynamoDbException e) {
            System.err.println(e.getMessage());
            return false;
        }
    }

    public static void createTable(DynamoDbClient ddb, String tableName) {
        DynamoDbWaiter dbWaiter = ddb.waiter();
        ArrayList<AttributeDefinition> attributeDefinitions = new ArrayList<>();

        // Define attributes.
        attributeDefinitions.add(AttributeDefinition.builder()
            .attributeName("year")
            .attributeType("N")
            .build());

        attributeDefinitions.add(AttributeDefinition.builder()
            .attributeName("title")
            .attributeType("S")
            .build());

        ArrayList<KeySchemaElement> tableKey = new ArrayList<>();
        KeySchemaElement key = KeySchemaElement.builder()
            .attributeName("year")
            .keyType(KeyType.HASH)
            .build();

        KeySchemaElement key2 = KeySchemaElement.builder()
            .attributeName("title")
            .keyType(KeyType.RANGE) // Sort
            .build();

        // Add KeySchemaElement objects to the list.
        tableKey.add(key);
        tableKey.add(key2);

        CreateTableRequest request = CreateTableRequest.builder()
            .keySchema(tableKey)
            .billingMode(BillingMode.PAY_PER_REQUEST) //  DynamoDB automatically
scales based on traffic.
            .attributeDefinitions(attributeDefinitions)
            .tableName(tableName)
            .build();
```

```
        try {
            CreateTableResponse response = ddb.createTable(request);
            DescribeTableRequest tableRequest = DescribeTableRequest.builder()
                .tableName(tableName)
                .build();

            // Wait until the Amazon DynamoDB table is created.
            WaiterResponse<DescribeTableResponse> waiterResponse = dbWaiter
                .waitUntilTableExists(tableRequest);
            waiterResponse.matched().response().ifPresent(System.out::println);
            String newTable = response.tableDescription().tableName();
            System.out.println("The " + newTable + " was successfully created.");

        } catch (DynamoDbException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }

    public static void putRecordBatch(DynamoDbClient ddb) {
        String sqlStatement = "INSERT INTO MoviesPartiQBatch VALUE {'year':?,
'title' : ?, 'info' : ?}";
        try {
            // Create three movies to add to the Amazon DynamoDB table.
            // Set data for Movie 1.
            List<AttributeValue> parameters = new ArrayList<>();

            AttributeValue att1 = AttributeValue.builder()
                .n("1977")
                .build();

            AttributeValue att2 = AttributeValue.builder()
                .s("Star Wars")
                .build();

            AttributeValue att3 = AttributeValue.builder()
                .s("No Information")
                .build();

            parameters.add(att1);
            parameters.add(att2);
            parameters.add(att3);
```

```
            BatchStatementRequest statementRequestMovie1 =
BatchStatementRequest.builder()
                .statement(sqlStatement)
                .parameters(parameters)
                .build();

            // Set data for Movie 2.
            List<AttributeValue> parametersMovie2 = new ArrayList<>();
            AttributeValue attMovie2 = AttributeValue.builder()
                .n("1939")
                .build();

            AttributeValue attMovie2A = AttributeValue.builder()
                .s("Wizard of Oz")
                .build();

            AttributeValue attMovie2B = AttributeValue.builder()
                .s("No Information")
                .build();

            parametersMovie2.add(attMovie2);
            parametersMovie2.add(attMovie2A);
            parametersMovie2.add(attMovie2B);

            BatchStatementRequest statementRequestMovie2 =
BatchStatementRequest.builder()
                .statement(sqlStatement)
                .parameters(parametersMovie2)
                .build();

            // Set data for Movie 3.
            List<AttributeValue> parametersMovie3 = new ArrayList<>();
            AttributeValue attMovie3 = AttributeValue.builder()
                .n(String.valueOf("2022"))
                .build();

            AttributeValue attMovie3A = AttributeValue.builder()
                .s("My Movie 3")
                .build();

            AttributeValue attMovie3B = AttributeValue.builder()
                .s("No Information")
                .build();
```

```
            parametersMovie3.add(attMovie3);
            parametersMovie3.add(attMovie3A);
            parametersMovie3.add(attMovie3B);

            BatchStatementRequest statementRequestMovie3 =
 BatchStatementRequest.builder()
                .statement(sqlStatement)
                .parameters(parametersMovie3)
                .build();

            // Add all three movies to the list.
            List<BatchStatementRequest> myBatchStatementList = new ArrayList<>();
            myBatchStatementList.add(statementRequestMovie1);
            myBatchStatementList.add(statementRequestMovie2);
            myBatchStatementList.add(statementRequestMovie3);

            BatchExecuteStatementRequest batchRequest =
 BatchExecuteStatementRequest.builder()
                .statements(myBatchStatementList)
                .build();

            BatchExecuteStatementResponse response =
 ddb.batchExecuteStatement(batchRequest);
            System.out.println("ExecuteStatement successful: " +
 response.toString());
            System.out.println("Added new movies using a batch command.");

        } catch (DynamoDbException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }

    public static void updateTableItemBatch(DynamoDbClient ddb) {
        String sqlStatement = "UPDATE MoviesPartiQBatch SET info = 'directors\":
[\"Merian C. Cooper\",\"Ernest B. Schoedsack' where year=? and title=?";
        List<AttributeValue> parametersRec1 = new ArrayList<>();

        // Update three records.
        AttributeValue att1 = AttributeValue.builder()
            .n(String.valueOf("2022"))
            .build();

        AttributeValue att2 = AttributeValue.builder()
```

```
            .s("My Movie 1")
            .build();

        parametersRec1.add(att1);
        parametersRec1.add(att2);

        BatchStatementRequest statementRequestRec1 =
BatchStatementRequest.builder()
            .statement(sqlStatement)
            .parameters(parametersRec1)
            .build();

        // Update record 2.
        List<AttributeValue> parametersRec2 = new ArrayList<>();
        AttributeValue attRec2 = AttributeValue.builder()
            .n(String.valueOf("2022"))
            .build();

        AttributeValue attRec2a = AttributeValue.builder()
            .s("My Movie 2")
            .build();

        parametersRec2.add(attRec2);
        parametersRec2.add(attRec2a);
        BatchStatementRequest statementRequestRec2 =
BatchStatementRequest.builder()
            .statement(sqlStatement)
            .parameters(parametersRec2)
            .build();

        // Update record 3.
        List<AttributeValue> parametersRec3 = new ArrayList<>();
        AttributeValue attRec3 = AttributeValue.builder()
            .n(String.valueOf("2022"))
            .build();

        AttributeValue attRec3a = AttributeValue.builder()
            .s("My Movie 3")
            .build();

        parametersRec3.add(attRec3);
        parametersRec3.add(attRec3a);
        BatchStatementRequest statementRequestRec3 =
BatchStatementRequest.builder()
```

```
            .statement(sqlStatement)
            .parameters(parametersRec3)
            .build();

        // Add all three movies to the list.
        List<BatchStatementRequest> myBatchStatementList = new ArrayList<>();
        myBatchStatementList.add(statementRequestRec1);
        myBatchStatementList.add(statementRequestRec2);
        myBatchStatementList.add(statementRequestRec3);

        BatchExecuteStatementRequest batchRequest =
BatchExecuteStatementRequest.builder()
            .statements(myBatchStatementList)
            .build();

        try {
            BatchExecuteStatementResponse response =
ddb.batchExecuteStatement(batchRequest);
            System.out.println("ExecuteStatement successful: " +
response.toString());
            System.out.println("Updated three movies using a batch command.");

        } catch (DynamoDbException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
        System.out.println("Item was updated!");
    }

    public static void deleteItemBatch(DynamoDbClient ddb) {
        String sqlStatement = "DELETE FROM MoviesPartiQBatch WHERE year = ? and
title=?";
        List<AttributeValue> parametersRec1 = new ArrayList<>();

        // Specify three records to delete.
        AttributeValue att1 = AttributeValue.builder()
            .n(String.valueOf("2022"))
            .build();

        AttributeValue att2 = AttributeValue.builder()
            .s("My Movie 1")
            .build();

        parametersRec1.add(att1);
```

```
        parametersRec1.add(att2);

        BatchStatementRequest statementRequestRec1 =
BatchStatementRequest.builder()
            .statement(sqlStatement)
            .parameters(parametersRec1)
            .build();

        // Specify record 2.
        List<AttributeValue> parametersRec2 = new ArrayList<>();
        AttributeValue attRec2 = AttributeValue.builder()
            .n(String.valueOf("2022"))
            .build();

        AttributeValue attRec2a = AttributeValue.builder()
            .s("My Movie 2")
            .build();

        parametersRec2.add(attRec2);
        parametersRec2.add(attRec2a);
        BatchStatementRequest statementRequestRec2 =
BatchStatementRequest.builder()
            .statement(sqlStatement)
            .parameters(parametersRec2)
            .build();

        // Specify record 3.
        List<AttributeValue> parametersRec3 = new ArrayList<>();
        AttributeValue attRec3 = AttributeValue.builder()
            .n(String.valueOf("2022"))
            .build();

        AttributeValue attRec3a = AttributeValue.builder()
            .s("My Movie 3")
            .build();

        parametersRec3.add(attRec3);
        parametersRec3.add(attRec3a);

        BatchStatementRequest statementRequestRec3 =
BatchStatementRequest.builder()
            .statement(sqlStatement)
            .parameters(parametersRec3)
            .build();
```

```
        // Add all three movies to the list.
        List<BatchStatementRequest> myBatchStatementList = new ArrayList<>();
        myBatchStatementList.add(statementRequestRec1);
        myBatchStatementList.add(statementRequestRec2);
        myBatchStatementList.add(statementRequestRec3);

        BatchExecuteStatementRequest batchRequest =
BatchExecuteStatementRequest.builder()
            .statements(myBatchStatementList)
            .build();

        try {
            ddb.batchExecuteStatement(batchRequest);
            System.out.println("Deleted three movies using a batch command.");

        } catch (DynamoDbException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }

    public static void deleteDynamoDBTable(DynamoDbClient ddb, String tableName)
{
        DeleteTableRequest request = DeleteTableRequest.builder()
            .tableName(tableName)
            .build();

        try {
            ddb.deleteTable(request);

        } catch (DynamoDbException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
        System.out.println(tableName + " was successfully deleted!");
    }

    private static ExecuteStatementResponse
executeStatementRequest(DynamoDbClient ddb, String statement,

List<AttributeValue> parameters) {
        ExecuteStatementRequest request = ExecuteStatementRequest.builder()
            .statement(statement)
```

```
            .parameters(parameters)
            .build();

        return ddb.executeStatement(request);
    }
}
```

- For API details, see BatchExecuteStatement in *AWS SDK for Java 2.x API Reference*.

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Execute batch PartiQL statements.

```
import {
  BillingMode,
  CreateTableCommand,
  DeleteTableCommand,
  DescribeTableCommand,
  DynamoDBClient,
  waitUntilTableExists,
} from "@aws-sdk/client-dynamodb";
import {
  DynamoDBDocumentClient,
  BatchExecuteStatementCommand,
} from "@aws-sdk/lib-dynamodb";
import { ScenarioInput } from "@aws-doc-sdk-examples/lib/scenario";

const client = new DynamoDBClient({});
const docClient = DynamoDBDocumentClient.from(client);

const log = (msg) => console.log(`[SCENARIO] ${msg}`);
const tableName = "Cities";
```

```
export const main = async (confirmAll = false) => {
  /**
   * Delete table if it exists.
   */
  try {
    await client.send(new DescribeTableCommand({ TableName: tableName }));
    // If no error was thrown, the table exists.
    const input = new ScenarioInput(
      "deleteTable",
      `A table named ${tableName} already exists. If you choose not to delete
this table, the scenario cannot continue. Delete it?`,
      { type: "confirm", confirmAll },
    );
    const deleteTable = await input.handle({}, { confirmAll });
    if (deleteTable) {
      await client.send(new DeleteTableCommand({ tableName }));
    } else {
      console.warn(
        "Scenario could not run. Either delete ${tableName} or provide a unique
 table name.",
      );
      return;
    }
  } catch (caught) {
    if (
      caught instanceof Error &&
      caught.name === "ResourceNotFoundException"
    ) {
      // Do nothing. This means the table is not there.
    } else {
      throw caught;
    }
  }

  /**
   * Create a table.
   */

  log("Creating a table.");
  const createTableCommand = new CreateTableCommand({
    TableName: tableName,
    // This example performs a large write to the database.
    // Set the billing mode to PAY_PER_REQUEST to
    // avoid throttling the large write.
```

```
    BillingMode: BillingMode.PAY_PER_REQUEST,
    // Define the attributes that are necessary for the key schema.
    AttributeDefinitions: [
      {
        AttributeName: "name",
        // 'S' is a data type descriptor that represents a number type.
        // For a list of all data type descriptors, see the following link.
        // https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/
Programming.LowLevelAPI.html#Programming.LowLevelAPI.DataTypeDescriptors
        AttributeType: "S",
      },
    ],
    // The KeySchema defines the primary key. The primary key can be
    // a partition key, or a combination of a partition key and a sort key.
    // Key schema design is important. For more info, see
    // https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/best-
practices.html
    KeySchema: [{ AttributeName: "name", KeyType: "HASH" }],
  });
  await client.send(createTableCommand);
  log(`Table created: ${tableName}.`);

  /**
   * Wait until the table is active.
   */

  // This polls with DescribeTableCommand until the requested table is 'ACTIVE'.
  // You can't write to a table before it's active.
  log("Waiting for the table to be active.");
  await waitUntilTableExists({ client }, { TableName: tableName });
  log("Table active.");

  /**
   * Insert items.
   */

  log("Inserting cities into the table.");
  const addItemsStatementCommand = new BatchExecuteStatementCommand({
    // https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/ql-
reference.insert.html
    Statements: [
      {
        Statement: `INSERT INTO ${tableName} value {'name':?, 'population':?}`,
        Parameters: ["Alachua", 10712],
```

```
      },
      {
        Statement: `INSERT INTO ${tableName} value {'name':?, 'population':?}`,
        Parameters: ["High Springs", 6415],
      },
    ],
  });
  await docClient.send(addItemsStatementCommand);
  log("Cities inserted.");

  /**
   * Select items.
   */

  log("Selecting cities from the table.");
  const selectItemsStatementCommand = new BatchExecuteStatementCommand({
    // https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/ql-
reference.select.html
    Statements: [
      {
        Statement: `SELECT * FROM ${tableName} WHERE name=?`,
        Parameters: ["Alachua"],
      },
      {
        Statement: `SELECT * FROM ${tableName} WHERE name=?`,
        Parameters: ["High Springs"],
      },
    ],
  });
  const selectItemResponse = await docClient.send(selectItemsStatementCommand);
  log(
    `Got cities: ${selectItemResponse.Responses.map(
      (r) => `${r.Item.name} (${r.Item.population})`,
    ).join(", ")}`,
  );

  /**
   * Update items.
   */

  log("Modifying the populations.");
  const updateItemStatementCommand = new BatchExecuteStatementCommand({
    // https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/ql-
reference.update.html
```

```
    Statements: [
      {
        Statement: `UPDATE ${tableName} SET population=? WHERE name=?`,
        Parameters: [10, "Alachua"],
      },
      {
        Statement: `UPDATE ${tableName} SET population=? WHERE name=?`,
        Parameters: [5, "High Springs"],
      },
    ],
  });
  await docClient.send(updateItemStatementCommand);
  log("Updated cities.");

  /**
   * Delete the items.
   */

  log("Deleting the cities.");
  const deleteItemStatementCommand = new BatchExecuteStatementCommand({
    // https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/ql-
reference.delete.html
    Statements: [
      {
        Statement: `DELETE FROM ${tableName} WHERE name=?`,
        Parameters: ["Alachua"],
      },
      {
        Statement: `DELETE FROM ${tableName} WHERE name=?`,
        Parameters: ["High Springs"],
      },
    ],
  });
  await docClient.send(deleteItemStatementCommand);
  log("Cities deleted.");

  /**
   * Delete the table.
   */

  log("Deleting the table.");
  const deleteTableCommand = new DeleteTableCommand({ TableName: tableName });
  await client.send(deleteTableCommand);
  log("Table deleted.");
```

```
};
```

- For API details, see BatchExecuteStatement in *AWS SDK for JavaScript API Reference*.

Kotlin

SDK for Kotlin

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
suspend fun main() {
    val ddb = DynamoDbClient.fromEnvironment { region = "us-east-1" }
    val tableName = "MoviesPartiQBatch"
    println("Creating an Amazon DynamoDB table named $tableName with a key named
 id and a sort key named title.")
    createTablePartiQLBatch(ddb, tableName, "year")
    putRecordBatch(ddb)
    updateTableItemBatchBatch(ddb)
    deleteItemsBatch(ddb)
    deleteTablePartiQLBatch(tableName)
}

suspend fun createTablePartiQLBatch(
    ddb: DynamoDbClient,
    tableNameVal: String,
    key: String,
) {
    val attDef =
        AttributeDefinition {
            attributeName = key
            attributeType = ScalarAttributeType.N
        }

    val attDef1 =
        AttributeDefinition {
            attributeName = "title"
            attributeType = ScalarAttributeType.S
```

```
        }

    val keySchemaVal =
        KeySchemaElement {
            attributeName = key
            keyType = KeyType.Hash
        }

    val keySchemaVal1 =
        KeySchemaElement {
            attributeName = "title"
            keyType = KeyType.Range
        }

    val request =
        CreateTableRequest {
            attributeDefinitions = listOf(attDef, attDef1)
            keySchema = listOf(keySchemaVal, keySchemaVal1)
            billingMode = BillingMode.PayPerRequest
            tableName = tableNameVal
        }

    val response = ddb.createTable(request)
    ddb.waitUntilTableExists {
        // suspend call
        tableName = tableNameVal
    }
    println("The table was successfully created
 ${response.tableDescription?.tableArn}")
}

suspend fun putRecordBatch(ddb: DynamoDbClient) {
    val sqlStatement = "INSERT INTO MoviesPartiQBatch VALUE {'year':?,
 'title' : ?, 'info' : ?}"

    // Create three movies to add to the Amazon DynamoDB table.
    val parametersMovie1 = mutableListOf<AttributeValue>()
    parametersMovie1.add(AttributeValue.N("2022"))
    parametersMovie1.add(AttributeValue.S("My Movie 1"))
    parametersMovie1.add(AttributeValue.S("No Information"))

    val statementRequestMovie1 =
        BatchStatementRequest {
            statement = sqlStatement
```

```
            parameters = parametersMovie1
        }

    // Set data for Movie 2.
    val parametersMovie2 = mutableListOf<AttributeValue>()
    parametersMovie2.add(AttributeValue.N("2022"))
    parametersMovie2.add(AttributeValue.S("My Movie 2"))
    parametersMovie2.add(AttributeValue.S("No Information"))

    val statementRequestMovie2 =
        BatchStatementRequest {
            statement = sqlStatement
            parameters = parametersMovie2
        }

    // Set data for Movie 3.
    val parametersMovie3 = mutableListOf<AttributeValue>()
    parametersMovie3.add(AttributeValue.N("2022"))
    parametersMovie3.add(AttributeValue.S("My Movie 3"))
    parametersMovie3.add(AttributeValue.S("No Information"))

    val statementRequestMovie3 =
        BatchStatementRequest {
            statement = sqlStatement
            parameters = parametersMovie3
        }

    // Add all three movies to the list.
    val myBatchStatementList = mutableListOf<BatchStatementRequest>()
    myBatchStatementList.add(statementRequestMovie1)
    myBatchStatementList.add(statementRequestMovie2)
    myBatchStatementList.add(statementRequestMovie3)

    val batchRequest =
        BatchExecuteStatementRequest {
            statements = myBatchStatementList
        }
    val response = ddb.batchExecuteStatement(batchRequest)
    println("ExecuteStatement successful: " + response.toString())
    println("Added new movies using a batch command.")
}

suspend fun updateTableItemBatchBatch(ddb: DynamoDbClient) {
    val sqlStatement =
```

```
        "UPDATE MoviesPartiQBatch SET info = 'directors\":[\"Merian C. Cooper\",
\"Ernest B. Schoedsack' where year=? and title=?"
    val parametersRec1 = mutableListOf<AttributeValue>()
    parametersRec1.add(AttributeValue.N("2022"))
    parametersRec1.add(AttributeValue.S("My Movie 1"))
    val statementRequestRec1 =
        BatchStatementRequest {
            statement = sqlStatement
            parameters = parametersRec1
        }

    // Update record 2.
    val parametersRec2 = mutableListOf<AttributeValue>()
    parametersRec2.add(AttributeValue.N("2022"))
    parametersRec2.add(AttributeValue.S("My Movie 2"))
    val statementRequestRec2 =
        BatchStatementRequest {
            statement = sqlStatement
            parameters = parametersRec2
        }

    // Update record 3.
    val parametersRec3 = mutableListOf<AttributeValue>()
    parametersRec3.add(AttributeValue.N("2022"))
    parametersRec3.add(AttributeValue.S("My Movie 3"))
    val statementRequestRec3 =
        BatchStatementRequest {
            statement = sqlStatement
            parameters = parametersRec3
        }

    // Add all three movies to the list.
    val myBatchStatementList = mutableListOf<BatchStatementRequest>()
    myBatchStatementList.add(statementRequestRec1)
    myBatchStatementList.add(statementRequestRec2)
    myBatchStatementList.add(statementRequestRec3)

    val batchRequest =
        BatchExecuteStatementRequest {
            statements = myBatchStatementList
        }

    val response = ddb.batchExecuteStatement(batchRequest)
    println("ExecuteStatement successful: $response")
```

```
    println("Updated three movies using a batch command.")
    println("Items were updated!")
}

suspend fun deleteItemsBatch(ddb: DynamoDbClient) {
    // Specify three records to delete.
    val sqlStatement = "DELETE FROM MoviesPartiQBatch WHERE year = ? and title=?"
    val parametersRec1 = mutableListOf<AttributeValue>()
    parametersRec1.add(AttributeValue.N("2022"))
    parametersRec1.add(AttributeValue.S("My Movie 1"))

    val statementRequestRec1 =
        BatchStatementRequest {
            statement = sqlStatement
            parameters = parametersRec1
        }

    // Specify record 2.
    val parametersRec2 = mutableListOf<AttributeValue>()
    parametersRec2.add(AttributeValue.N("2022"))
    parametersRec2.add(AttributeValue.S("My Movie 2"))
    val statementRequestRec2 =
        BatchStatementRequest {
            statement = sqlStatement
            parameters = parametersRec2
        }

    // Specify record 3.
    val parametersRec3 = mutableListOf<AttributeValue>()
    parametersRec3.add(AttributeValue.N("2022"))
    parametersRec3.add(AttributeValue.S("My Movie 3"))
    val statementRequestRec3 =
        BatchStatementRequest {
            statement = sqlStatement
            parameters = parametersRec3
        }

    // Add all three movies to the list.
    val myBatchStatementList = mutableListOf<BatchStatementRequest>()
    myBatchStatementList.add(statementRequestRec1)
    myBatchStatementList.add(statementRequestRec2)
    myBatchStatementList.add(statementRequestRec3)

    val batchRequest =
```

```
        BatchExecuteStatementRequest {
            statements = myBatchStatementList
        }

    ddb.batchExecuteStatement(batchRequest)
    println("Deleted three movies using a batch command.")
}

suspend fun deleteTablePartiQLBatch(tableNameVal: String) {
    val request =
        DeleteTableRequest {
            tableName = tableNameVal
        }

    DynamoDbClient { region = "us-east-1" }.use { ddb ->
        ddb.deleteTable(request)
        println("$tableNameVal was deleted")
    }
}
```

- For API details, see BatchExecuteStatement in *AWS SDK for Kotlin API reference*.

PHP

SDK for PHP

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
namespace DynamoDb\PartiQL_Basics;

use Aws\DynamoDb\Marshaler;
use DynamoDb;
use DynamoDb\DynamoDBAttribute;

use function AwsUtilities\loadMovieData;
use function AwsUtilities\testable_readline;
```

```
class GettingStartedWithPartiQLBatch
{
    public function run()
    {
        echo("\n");
        echo("--------------------------------------\n");
        print("Welcome to the Amazon DynamoDB - PartiQL getting started demo
 using PHP!\n");
        echo("--------------------------------------\n");

        $uuid = uniqid();
        $service = new DynamoDb\DynamoDBService();

        $tableName = "partiql_demo_table_$uuid";
        $service->createTable(
            $tableName,
            [
                new DynamoDBAttribute('year', 'N', 'HASH'),
                new DynamoDBAttribute('title', 'S', 'RANGE')
            ]
        );

        echo "Waiting for table...";
        $service->dynamoDbClient->waitUntil("TableExists", ['TableName' =>
 $tableName]);
        echo "table $tableName found!\n";

        echo "What's the name of the last movie you watched?\n";
        while (empty($movieName)) {
            $movieName = testable_readline("Movie name: ");
        }
        echo "And what year was it released?\n";
        $movieYear = "year";
        while (!is_numeric($movieYear) || intval($movieYear) != $movieYear) {
            $movieYear = testable_readline("Year released: ");
        }
        $key = [
            'Item' => [
                'year' => [
                    'N' => "$movieYear",
                ],
                'title' => [
                    'S' => $movieName,
                ],
```

```
            ],
        ];
        list($statement, $parameters) = $service-
>buildStatementAndParameters("INSERT", $tableName, $key);
        $service->insertItemByPartiQLBatch($statement, $parameters);

        echo "How would you rate the movie from 1-10?\n";
        $rating = 0;
        while (!is_numeric($rating) || intval($rating) != $rating || $rating < 1
 || $rating > 10) {
            $rating = testable_readline("Rating (1-10): ");
        }
        echo "What was the movie about?\n";
        while (empty($plot)) {
            $plot = testable_readline("Plot summary: ");
        }
        $attributes = [
            new DynamoDBAttribute('rating', 'N', 'HASH', $rating),
            new DynamoDBAttribute('plot', 'S', 'RANGE', $plot),
        ];

        list($statement, $parameters) = $service-
>buildStatementAndParameters("UPDATE", $tableName, $key, $attributes);
        $service->updateItemByPartiQLBatch($statement, $parameters);
        echo "Movie added and updated.\n";

        $batch = json_decode(loadMovieData());

        $service->writeBatch($tableName, $batch);

        $movie = $service->getItemByPartiQLBatch($tableName, [$key]);
        echo "\nThe movie {$movie['Responses'][0]['Item']['title']['S']}
        was released in {$movie['Responses'][0]['Item']['year']['N']}.\n";
        echo "What rating would you like to give {$movie['Responses'][0]['Item']
['title']['S']}?\n";
        $rating = 0;
        while (!is_numeric($rating) || intval($rating) != $rating || $rating < 1
 || $rating > 10) {
            $rating = testable_readline("Rating (1-10): ");
        }
        $attributes = [
            new DynamoDBAttribute('rating', 'N', 'HASH', $rating),
            new DynamoDBAttribute('plot', 'S', 'RANGE', $plot)
        ];
```

```
        list($statement, $parameters) = $service-
>buildStatementAndParameters("UPDATE", $tableName, $key, $attributes);
        $service->updateItemByPartiQLBatch($statement, $parameters);

        $movie = $service->getItemByPartiQLBatch($tableName, [$key]);
        echo "Okay, you have rated {$movie['Responses'][0]['Item']['title']
['S']}
        as a {$movie['Responses'][0]['Item']['rating']['N']}\n";

        $service->deleteItemByPartiQLBatch($statement, $parameters);
        echo "But, bad news, this was a trap. That movie has now been deleted
 because of your rating...harsh.\n";

        echo "That's okay though. The book was better. Now, for something
 lighter, in what year were you born?\n";
        $birthYear = "not a number";
        while (!is_numeric($birthYear) || $birthYear >= date("Y")) {
            $birthYear = testable_readline("Birth year: ");
        }
        $birthKey = [
            'Key' => [
                'year' => [
                    'N' => "$birthYear",
                ],
            ],
        ];
        $result = $service->query($tableName, $birthKey);
        $marshal = new Marshaler();
        echo "Here are the movies in our collection released the year you were
 born:\n";
        $oops = "Oops! There were no movies released in that year (that we know
 of).\n";
        $display = "";
        foreach ($result['Items'] as $movie) {
            $movie = $marshal->unmarshalItem($movie);
            $display .= $movie['title'] . "\n";
        }
        echo ($display) ?: $oops;

        $yearsKey = [
            'Key' => [
                'year' => [
                    'N' => [
                        'minRange' => 1990,
```

```
                        'maxRange' => 1999,
                    ],
                ],
            ],
        ];
        $filter = "year between 1990 and 1999";
        echo "\nHere's a list of all the movies released in the 90s:\n";
        $result = $service->scan($tableName, $yearsKey, $filter);
        foreach ($result['Items'] as $movie) {
            $movie = $marshal->unmarshalItem($movie);
            echo $movie['title'] . "\n";
        }

        echo "\nCleaning up this demo by deleting table $tableName...\n";
        $service->deleteTable($tableName);
    }
}

    public function insertItemByPartiQLBatch(string $statement, array
 $parameters)
    {
        $this->dynamoDbClient->batchExecuteStatement([
            'Statements' => [
                [
                    'Statement' => "$statement",
                    'Parameters' => $parameters,
                ],
            ],
        ]);
    }

    public function getItemByPartiQLBatch(string $tableName, array $keys): Result
    {
        $statements = [];
        foreach ($keys as $key) {
            list($statement, $parameters) = $this-
>buildStatementAndParameters("SELECT", $tableName, $key['Item']);
            $statements[] = [
                'Statement' => "$statement",
                'Parameters' => $parameters,
            ];
        }

        return $this->dynamoDbClient->batchExecuteStatement([
```

```
            'Statements' => $statements,
        ]);
    }

    public function updateItemByPartiQLBatch(string $statement, array
$parameters)
    {
        $this->dynamoDbClient->batchExecuteStatement([
            'Statements' => [
                [
                    'Statement' => "$statement",
                    'Parameters' => $parameters,
                ],
            ],
        ]);
    }

    public function deleteItemByPartiQLBatch(string $statement, array
$parameters)
    {
        $this->dynamoDbClient->batchExecuteStatement([
            'Statements' => [
                [
                    'Statement' => "$statement",
                    'Parameters' => $parameters,
                ],
            ],
        ]);
    }
```

- For API details, see BatchExecuteStatement in *AWS SDK for PHP API Reference*.

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Create a class that can run batches of PartiQL statements.

```
from datetime import datetime
from decimal import Decimal
import logging
from pprint import pprint

import boto3
from botocore.exceptions import ClientError

from scaffold import Scaffold

logger = logging.getLogger(__name__)

class PartiQLBatchWrapper:
    """
    Encapsulates a DynamoDB resource to run PartiQL statements.
    """

    def __init__(self, dyn_resource):
        """
        :param dyn_resource: A Boto3 DynamoDB resource.
        """
        self.dyn_resource = dyn_resource


    def run_partiql(self, statements, param_list):
        """
        Runs a PartiQL statement. A Boto3 resource is used even though
        `execute_statement` is called on the underlying `client` object because
 the
        resource transforms input and output from plain old Python objects
 (POPOs) to
        the DynamoDB format. If you create the client directly, you must do these
        transforms yourself.

        :param statements: The batch of PartiQL statements.
        :param param_list: The batch of PartiQL parameters that are associated
 with
                           each statement. This list must be in the same order as
 the
                           statements.
        :return: The responses returned from running the statements, if any.
        """
```

```
        try:
            output = self.dyn_resource.meta.client.batch_execute_statement(
                Statements=[
                    {"Statement": statement, "Parameters": params}
                    for statement, params in zip(statements, param_list)
                ]
            )
        except ClientError as err:
            if err.response["Error"]["Code"] == "ResourceNotFoundException":
                logger.error(
                    "Couldn't execute batch of PartiQL statements because the
 table "
                    "does not exist."
                )
            else:
                logger.error(
                    "Couldn't execute batch of PartiQL statements. Here's why:
 %s: %s",
                    err.response["Error"]["Code"],
                    err.response["Error"]["Message"],
                )
            raise
        else:
            return output
```

Run a scenario that creates a table and runs PartiQL queries in batches.

```
def run_scenario(scaffold, wrapper, table_name):
    logging.basicConfig(level=logging.INFO, format="%(levelname)s: %(message)s")

    print("-" * 88)
    print("Welcome to the Amazon DynamoDB PartiQL batch statement demo.")
    print("-" * 88)

    print(f"Creating table '{table_name}' for the demo...")
    scaffold.create_table(table_name)
    print("-" * 88)

    movie_data = [
```

```
        {
            "title": f"House PartiQL",
            "year": datetime.now().year - 5,
            "info": {
                "plot": "Wacky high jinks result from querying a mysterious
database.",
                "rating": Decimal("8.5"),
            },
        },
        {
            "title": f"House PartiQL 2",
            "year": datetime.now().year - 3,
            "info": {
                "plot": "Moderate high jinks result from querying another
mysterious database.",
                "rating": Decimal("6.5"),
            },
        },
        {
            "title": f"House PartiQL 3",
            "year": datetime.now().year - 1,
            "info": {
                "plot": "Tepid high jinks result from querying yet another
mysterious database.",
                "rating": Decimal("2.5"),
            },
        },
    ]

    print(f"Inserting a batch of movies into table '{table_name}.")
    statements = [
        f'INSERT INTO "{table_name}" ' f"VALUE {{'title': ?, 'year': ?,
'info': ?}}"
    ] * len(movie_data)
    params = [list(movie.values()) for movie in movie_data]
    wrapper.run_partiql(statements, params)
    print("Success!")
    print("-" * 88)

    print(f"Getting data for a batch of movies.")
    statements = [f'SELECT * FROM "{table_name}" WHERE title=? AND year=?'] *
len(
        movie_data
    )
```

```
    params = [[movie["title"], movie["year"]] for movie in movie_data]
    output = wrapper.run_partiql(statements, params)
    for item in output["Responses"]:
        print(f"\n{item['Item']['title']}, {item['Item']['year']}")
        pprint(item["Item"])
    print("-" * 88)

    ratings = [Decimal("7.7"), Decimal("5.5"), Decimal("1.3")]
    print(f"Updating a batch of movies with new ratings.")
    statements = [
        f'UPDATE "{table_name}" SET info.rating=? ' f"WHERE title=? AND year=?"
    ] * len(movie_data)
    params = [
        [rating, movie["title"], movie["year"]]
        for rating, movie in zip(ratings, movie_data)
    ]
    wrapper.run_partiql(statements, params)
    print("Success!")
    print("-" * 88)

    print(f"Getting projected data from the table to verify our update.")
    output = wrapper.dyn_resource.meta.client.execute_statement(
        Statement=f'SELECT title, info.rating FROM "{table_name}"'
    )
    pprint(output["Items"])
    print("-" * 88)

    print(f"Deleting a batch of movies from the table.")
    statements = [f'DELETE FROM "{table_name}" WHERE title=? AND year=?'] * len(
        movie_data
    )
    params = [[movie["title"], movie["year"]] for movie in movie_data]
    wrapper.run_partiql(statements, params)
    print("Success!")
    print("-" * 88)

    print(f"Deleting table '{table_name}'...")
    scaffold.delete_table()
    print("-" * 88)

    print("\nThanks for watching!")
    print("-" * 88)
```

```
if __name__ == "__main__":
    try:
        dyn_res = boto3.resource("dynamodb")
        scaffold = Scaffold(dyn_res)
        movies = PartiQLBatchWrapper(dyn_res)
        run_scenario(scaffold, movies, "doc-example-table-partiql-movies")
    except Exception as e:
        print(f"Something went wrong with the demo! Here's what: {e}")
```

- For API details, see BatchExecuteStatement in *AWS SDK for Python (Boto3) API Reference*.

Ruby

SDK for Ruby

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Run a scenario that creates a table and runs batch PartiQL queries.

```
table_name = "doc-example-table-movies-partiql-#{rand(10**4)}"
scaffold = Scaffold.new(table_name)
sdk = DynamoDBPartiQLBatch.new(table_name)

new_step(1, 'Create a new DynamoDB table if none already exists.')
unless scaffold.exists?(table_name)
  puts("\nNo such table: #{table_name}. Creating it...")
  scaffold.create_table(table_name)
  print "Done!\n".green
end

new_step(2, 'Populate DynamoDB table with movie data.')
download_file = 'moviedata.json'
puts("Downloading movie database to #{download_file}...")
movie_data = scaffold.fetch_movie_data(download_file)
puts("Writing movie data from #{download_file} into your table...")
scaffold.write_batch(movie_data)
puts("Records added: #{movie_data.length}.")
```

```
  print "Done!\n".green

  new_step(3, 'Select a batch of items from the movies table.')
  puts "Let's select some popular movies for side-by-side comparison."
  response = sdk.batch_execute_select([['Mean Girls', 2004], ['Goodfellas',
 1977], ['The Prancing of the Lambs', 2005]])
  puts("Items selected: #{response['responses'].length}\n")
  print "\nDone!\n".green

  new_step(4, 'Delete a batch of items from the movies table.')
  sdk.batch_execute_write([['Mean Girls', 2004], ['Goodfellas', 1977], ['The
 Prancing of the Lambs', 2005]])
  print "\nDone!\n".green

  new_step(5, 'Delete the table.')
  return unless scaffold.exists?(table_name)

  scaffold.delete_table
end
```

- For API details, see BatchExecuteStatement in *AWS SDK for Ruby API Reference*.

For a complete list of AWS SDK developer guides and code examples, see Using DynamoDB with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Query a DynamoDB table using PartiQL and an AWS SDK

The following code examples show how to:

- Get an item by running a SELECT statement.
- Add an item by running an INSERT statement.
- Update an item by running an UPDATE statement.
- Delete an item by running a DELETE statement.

.NET

SDK for .NET

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
namespace PartiQL_Basics_Scenario
{
    public class PartiQLMethods
    {
        private static readonly AmazonDynamoDBClient Client = new
 AmazonDynamoDBClient();


        /// <summary>
        /// Inserts movies imported from a JSON file into the movie table by
        /// using an Amazon DynamoDB PartiQL INSERT statement.
        /// </summary>
        /// <param name="tableName">The name of the table where the movie
        /// information will be inserted.</param>
        /// <param name="movieFileName">The name of the JSON file that contains
        /// movie information.</param>
        /// <returns>A Boolean value that indicates the success or failure of
        /// the insert operation.</returns>
        public static async Task<bool> InsertMovies(string tableName, string
 movieFileName)
        {
            // Get the list of movies from the JSON file.
            var movies = ImportMovies(movieFileName);

            var success = false;

            if (movies is not null)
            {
                // Insert the movies in a batch using PartiQL. Because the
                // batch can contain a maximum of 25 items, insert 25 movies
                // at a time.
```

```
                string insertBatch = $"INSERT INTO {tableName} VALUE
{{'title': ?, 'year': ?}}";
                var statements = new List<BatchStatementRequest>();

                try
                {
                    for (var indexOffset = 0; indexOffset < 250; indexOffset +=
25)
                    {
                        for (var i = indexOffset; i < indexOffset + 25; i++)
                        {
                            statements.Add(new BatchStatementRequest
                            {
                                Statement = insertBatch,
                                Parameters = new List<AttributeValue>
                                {
                                    new AttributeValue { S = movies[i].Title },
                                    new AttributeValue { N =
movies[i].Year.ToString() },
                                },
                            });
                        }

                        var response = await
Client.BatchExecuteStatementAsync(new BatchExecuteStatementRequest
                        {
                            Statements = statements,
                        });

                        // Wait between batches for movies to be successfully
added.
                        System.Threading.Thread.Sleep(3000);

                        success = response.HttpStatusCode ==
System.Net.HttpStatusCode.OK;

                        // Clear the list of statements for the next batch.
                        statements.Clear();
                    }
                }
                catch (AmazonDynamoDBException ex)
                {
                    Console.WriteLine(ex.Message);
                }
```

```
            }

            return success;
        }

        /// <summary>
        /// Loads the contents of a JSON file into a list of movies to be
        /// added to the DynamoDB table.
        /// </summary>
        /// <param name="movieFileName">The full path to the JSON file.</param>
        /// <returns>A generic list of movie objects.</returns>
        public static List<Movie> ImportMovies(string movieFileName)
        {
            if (!File.Exists(movieFileName))
            {
                return null!;
            }

            using var sr = new StreamReader(movieFileName);
            string json = sr.ReadToEnd();
            var allMovies = JsonConvert.DeserializeObject<List<Movie>>(json);

            if (allMovies is not null)
            {
                // Return the first 250 entries.
                return allMovies.GetRange(0, 250);
            }
            else
            {
                return null!;
            }
        }



        /// <summary>
        /// Uses a PartiQL SELECT statement to retrieve a single movie from the
        /// movie database.
        /// </summary>
        /// <param name="tableName">The name of the movie table.</param>
        /// <param name="movieTitle">The title of the movie to retrieve.</param>
        /// <returns>A list of movie data. If no movie matches the supplied
        /// title, the list is empty.</returns>
```

```
        public static async Task<List<Dictionary<string, AttributeValue>>>
GetSingleMovie(string tableName, string movieTitle)
        {
            string selectSingle = $"SELECT * FROM {tableName} WHERE title = ?";
            var parameters = new List<AttributeValue>
            {
                new AttributeValue { S = movieTitle },
            };

            var response = await Client.ExecuteStatementAsync(new
ExecuteStatementRequest
            {
                Statement = selectSingle,
                Parameters = parameters,
            });

            return response.Items;
        }



        /// <summary>
        /// Retrieve multiple movies by year using a SELECT statement.
        /// </summary>
        /// <param name="tableName">The name of the movie table.</param>
        /// <param name="year">The year the movies were released.</param>
        /// <returns></returns>
        public static async Task<List<Dictionary<string, AttributeValue>>>
GetMovies(string tableName, int year)
        {
            string selectSingle = $"SELECT * FROM {tableName} WHERE year = ?";
            var parameters = new List<AttributeValue>
            {
                new AttributeValue { N = year.ToString() },
            };

            var response = await Client.ExecuteStatementAsync(new
ExecuteStatementRequest
            {
                Statement = selectSingle,
                Parameters = parameters,
            });

            return response.Items;
```

```
        }


        /// <summary>
        /// Inserts a single movie into the movies table.
        /// </summary>
        /// <param name="tableName">The name of the table.</param>
        /// <param name="movieTitle">The title of the movie to insert.</param>
        /// <param name="year">The year that the movie was released.</param>
        /// <returns>A Boolean value that indicates the success or failure of
        /// the INSERT operation.</returns>
        public static async Task<bool> InsertSingleMovie(string tableName, string
movieTitle, int year)
        {
            string insertBatch = $"INSERT INTO {tableName} VALUE {{'title': ?,
'year': ?}}";

            var response = await Client.ExecuteStatementAsync(new
ExecuteStatementRequest
            {
                Statement = insertBatch,
                Parameters = new List<AttributeValue>
                {
                    new AttributeValue { S = movieTitle },
                    new AttributeValue { N = year.ToString() },
                },
            });

            return response.HttpStatusCode == System.Net.HttpStatusCode.OK;
        }



        /// <summary>
        /// Updates a single movie in the table, adding information for the
        /// producer.
        /// </summary>
        /// <param name="tableName">the name of the table.</param>
        /// <param name="producer">The name of the producer.</param>
        /// <param name="movieTitle">The movie title.</param>
        /// <param name="year">The year the movie was released.</param>
        /// <returns>A Boolean value that indicates the success of the
        /// UPDATE operation.</returns>
```

```
        public static async Task<bool> UpdateSingleMovie(string tableName, string
producer, string movieTitle, int year)
        {
            string insertSingle = $"UPDATE {tableName} SET Producer=? WHERE title
= ? AND year = ?";

            var response = await Client.ExecuteStatementAsync(new
ExecuteStatementRequest
            {
                Statement = insertSingle,
                Parameters = new List<AttributeValue>
                {
                    new AttributeValue { S = producer },
                    new AttributeValue { S = movieTitle },
                    new AttributeValue { N = year.ToString() },
                },
            });

            return response.HttpStatusCode == System.Net.HttpStatusCode.OK;
        }



        /// <summary>
        /// Deletes a single movie from the table.
        /// </summary>
        /// <param name="tableName">The name of the table.</param>
        /// <param name="movieTitle">The title of the movie to delete.</param>
        /// <param name="year">The year that the movie was released.</param>
        /// <returns>A Boolean value that indicates the success of the
        /// DELETE operation.</returns>
        public static async Task<bool> DeleteSingleMovie(string tableName, string
movieTitle, int year)
        {
            var deleteSingle = $"DELETE FROM {tableName} WHERE title = ? AND year
= ?";

            var response = await Client.ExecuteStatementAsync(new
ExecuteStatementRequest
            {
                Statement = deleteSingle,
                Parameters = new List<AttributeValue>
                {
                    new AttributeValue { S = movieTitle },
```

```
                    new AttributeValue { N = year.ToString() },
                },
            });

            return response.HttpStatusCode == System.Net.HttpStatusCode.OK;
        }


        /// <summary>
        /// Displays the list of movies returned from a database query.
        /// </summary>
        /// <param name="items">The list of movie information to display.</param>
        private static void DisplayMovies(List<Dictionary<string,
 AttributeValue>> items)
        {
            if (items.Count > 0)
            {
                Console.WriteLine($"Found {items.Count} movies.");
                items.ForEach(item =>
 Console.WriteLine($"{item["year"].N}\t{item["title"].S}"));
            }
            else
            {
                Console.WriteLine($"Didn't find a movie that matched the supplied
 criteria.");
            }
        }


    }
}




        /// <summary>
        /// Uses a PartiQL SELECT statement to retrieve a single movie from the
        /// movie database.
        /// </summary>
        /// <param name="tableName">The name of the movie table.</param>
        /// <param name="movieTitle">The title of the movie to retrieve.</param>
        /// <returns>A list of movie data. If no movie matches the supplied
        /// title, the list is empty.</returns>
        public static async Task<List<Dictionary<string, AttributeValue>>>
 GetSingleMovie(string tableName, string movieTitle)
```

```
        {
            string selectSingle = $"SELECT * FROM {tableName} WHERE title = ?";
            var parameters = new List<AttributeValue>
            {
                new AttributeValue { S = movieTitle },
            };

            var response = await Client.ExecuteStatementAsync(new
ExecuteStatementRequest
            {
                Statement = selectSingle,
                Parameters = parameters,
            });

            return response.Items;
        }



        /// <summary>
        /// Inserts a single movie into the movies table.
        /// </summary>
        /// <param name="tableName">The name of the table.</param>
        /// <param name="movieTitle">The title of the movie to insert.</param>
        /// <param name="year">The year that the movie was released.</param>
        /// <returns>A Boolean value that indicates the success or failure of
        /// the INSERT operation.</returns>
        public static async Task<bool> InsertSingleMovie(string tableName, string
movieTitle, int year)
        {
            string insertBatch = $"INSERT INTO {tableName} VALUE {{'title': ?,
'year': ?}}";

            var response = await Client.ExecuteStatementAsync(new
ExecuteStatementRequest
            {
                Statement = insertBatch,
                Parameters = new List<AttributeValue>
                {
                    new AttributeValue { S = movieTitle },
                    new AttributeValue { N = year.ToString() },
                },
            });
```

```
            return response.HttpStatusCode == System.Net.HttpStatusCode.OK;
        }



        /// <summary>
        /// Updates a single movie in the table, adding information for the
        /// producer.
        /// </summary>
        /// <param name="tableName">the name of the table.</param>
        /// <param name="producer">The name of the producer.</param>
        /// <param name="movieTitle">The movie title.</param>
        /// <param name="year">The year the movie was released.</param>
        /// <returns>A Boolean value that indicates the success of the
        /// UPDATE operation.</returns>
        public static async Task<bool> UpdateSingleMovie(string tableName, string
producer, string movieTitle, int year)
        {
            string insertSingle = $"UPDATE {tableName} SET Producer=? WHERE title
= ? AND year = ?";

            var response = await Client.ExecuteStatementAsync(new
ExecuteStatementRequest
            {
                Statement = insertSingle,
                Parameters = new List<AttributeValue>
                {
                    new AttributeValue { S = producer },
                    new AttributeValue { S = movieTitle },
                    new AttributeValue { N = year.ToString() },
                },
            });

            return response.HttpStatusCode == System.Net.HttpStatusCode.OK;
        }



        /// <summary>
        /// Deletes a single movie from the table.
        /// </summary>
        /// <param name="tableName">The name of the table.</param>
        /// <param name="movieTitle">The title of the movie to delete.</param>
        /// <param name="year">The year that the movie was released.</param>
```

```
        /// <returns>A Boolean value that indicates the success of the
        /// DELETE operation.</returns>
        public static async Task<bool> DeleteSingleMovie(string tableName, string
movieTitle, int year)
        {
            var deleteSingle = $"DELETE FROM {tableName} WHERE title = ? AND year
= ?";

            var response = await Client.ExecuteStatementAsync(new
ExecuteStatementRequest
            {
                Statement = deleteSingle,
                Parameters = new List<AttributeValue>
                {
                    new AttributeValue { S = movieTitle },
                    new AttributeValue { N = year.ToString() },
                },
            });

            return response.HttpStatusCode == System.Net.HttpStatusCode.OK;
        }
```

- For API details, see ExecuteStatement in *AWS SDK for .NET API Reference*.

C++

SDK for C++

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
        //  1. Create a table. (CreateTable)
        if (AwsDoc::DynamoDB::createMoviesDynamoDBTable(clientConfig)) {

            AwsDoc::DynamoDB::partiqlExecuteScenario(clientConfig);

            // 7. Delete the table. (DeleteTable)
```

```
            AwsDoc::DynamoDB::deleteMoviesDynamoDBTable(clientConfig);
        }

//! Scenario to modify and query a DynamoDB table using single PartiQL
 statements.
/*!
  \sa partiqlExecuteScenario()
  \param clientConfiguration: AWS client configuration.
  \return bool: Function succeeded.
 */
bool
AwsDoc::DynamoDB::partiqlExecuteScenario(
        const Aws::Client::ClientConfiguration &clientConfiguration) {
    Aws::DynamoDB::DynamoDBClient dynamoClient(clientConfiguration);

    // 2. Add a new movie using an "Insert" statement. (ExecuteStatement)
    Aws::String title;
    float rating;
    int year;
    Aws::String plot;
    {
        title = askQuestion(
                "Enter the title of a movie you want to add to the table: ");
        year = askQuestionForInt("What year was it released? ");
        rating = askQuestionForFloatRange("On a scale of 1 - 10, how do you rate
 it? ",
                                          1, 10);
        plot = askQuestion("Summarize the plot for me: ");

        Aws::DynamoDB::Model::ExecuteStatementRequest request;
        std::stringstream sqlStream;
        sqlStream << "INSERT INTO \"" << MOVIE_TABLE_NAME << "\" VALUE {'"
                  << TITLE_KEY << "': ?, '" << YEAR_KEY << "': ?, '"
                  << INFO_KEY << "': ?}";

        request.SetStatement(sqlStream.str());

        // Create the parameter attributes.
        Aws::Vector<Aws::DynamoDB::Model::AttributeValue> attributes;
        attributes.push_back(Aws::DynamoDB::Model::AttributeValue().SetS(title));
        attributes.push_back(Aws::DynamoDB::Model::AttributeValue().SetN(year));

        Aws::DynamoDB::Model::AttributeValue infoMapAttribute;
```

```
        std::shared_ptr<Aws::DynamoDB::Model::AttributeValue> ratingAttribute =
Aws::MakeShared<Aws::DynamoDB::Model::AttributeValue>(
                ALLOCATION_TAG.c_str());
        ratingAttribute->SetN(rating);
        infoMapAttribute.AddMEntry(RATING_KEY, ratingAttribute);

        std::shared_ptr<Aws::DynamoDB::Model::AttributeValue> plotAttribute =
Aws::MakeShared<Aws::DynamoDB::Model::AttributeValue>(
                ALLOCATION_TAG.c_str());
        plotAttribute->SetS(plot);
        infoMapAttribute.AddMEntry(PLOT_KEY, plotAttribute);
        attributes.push_back(infoMapAttribute);
        request.SetParameters(attributes);

        Aws::DynamoDB::Model::ExecuteStatementOutcome outcome =
dynamoClient.ExecuteStatement(
                request);

        if (!outcome.IsSuccess()) {
            std::cerr << "Failed to add a movie: " <<
outcome.GetError().GetMessage()
                      << std::endl;
            return false;
        }
    }

    std::cout << "\nAdded '" << title << "' to '" << MOVIE_TABLE_NAME << "'."
              << std::endl;

    //  3. Get the data for the movie using a "Select" statement.
(ExecuteStatement)
    {
        Aws::DynamoDB::Model::ExecuteStatementRequest request;
        std::stringstream sqlStream;
        sqlStream << "SELECT * FROM  \"" << MOVIE_TABLE_NAME << "\" WHERE "
                  << TITLE_KEY << "=? and " << YEAR_KEY << "=?";

        request.SetStatement(sqlStream.str());

        Aws::Vector<Aws::DynamoDB::Model::AttributeValue> attributes;
        attributes.push_back(Aws::DynamoDB::Model::AttributeValue().SetS(title));
        attributes.push_back(Aws::DynamoDB::Model::AttributeValue().SetN(year));
        request.SetParameters(attributes);
```

```
        Aws::DynamoDB::Model::ExecuteStatementOutcome outcome =
dynamoClient.ExecuteStatement(
                request);

        if (!outcome.IsSuccess()) {
            std::cerr << "Failed to retrieve movie information: "
                      << outcome.GetError().GetMessage() << std::endl;
            return false;
        }
        else {
            // Print the retrieved movie information.
            const Aws::DynamoDB::Model::ExecuteStatementResult &result =
outcome.GetResult();

            const Aws::Vector<Aws::Map<Aws::String,
Aws::DynamoDB::Model::AttributeValue>> &items = result.GetItems();

            if (items.size() == 1) {
                printMovieInfo(items[0]);
            }
            else {
                std::cerr << "Error: " << items.size() << " movies were
retrieved. "
                          << " There should be only one movie." << std::endl;
            }
        }
    }

    //  4. Update the data for the movie using an "Update" statement.
(ExecuteStatement)
    {
        rating = askQuestionForFloatRange(
                Aws::String("\nLet's update your movie.\nYou rated it  ") +
                std::to_string(rating)
                + ", what new rating would you give it? ", 1, 10);

        Aws::DynamoDB::Model::ExecuteStatementRequest request;
        std::stringstream sqlStream;
        sqlStream << "UPDATE \"" << MOVIE_TABLE_NAME << "\" SET "
                  << INFO_KEY << "." << RATING_KEY << "=? WHERE "
                  << TITLE_KEY << "=? AND " << YEAR_KEY << "=?";

        request.SetStatement(sqlStream.str());
```

```
        Aws::Vector<Aws::DynamoDB::Model::AttributeValue> attributes;

attributes.push_back(Aws::DynamoDB::Model::AttributeValue().SetN(rating));
        attributes.push_back(Aws::DynamoDB::Model::AttributeValue().SetS(title));
        attributes.push_back(Aws::DynamoDB::Model::AttributeValue().SetN(year));

        request.SetParameters(attributes);

        Aws::DynamoDB::Model::ExecuteStatementOutcome outcome =
dynamoClient.ExecuteStatement(
                request);

        if (!outcome.IsSuccess()) {
            std::cerr << "Failed to update a movie: "
                      << outcome.GetError().GetMessage();
            return false;
        }
    }

    std::cout << "\nUpdated '" << title << "' with new attributes:" << std::endl;

    //  5. Get the updated data for the movie using a "Select" statement.
(ExecuteStatement)
    {
        Aws::DynamoDB::Model::ExecuteStatementRequest request;
        std::stringstream sqlStream;
        sqlStream << "SELECT * FROM  \"" << MOVIE_TABLE_NAME << "\" WHERE "
                  << TITLE_KEY << "=? and " << YEAR_KEY << "=?";

        request.SetStatement(sqlStream.str());

        Aws::Vector<Aws::DynamoDB::Model::AttributeValue> attributes;
        attributes.push_back(Aws::DynamoDB::Model::AttributeValue().SetS(title));
        attributes.push_back(Aws::DynamoDB::Model::AttributeValue().SetN(year));
        request.SetParameters(attributes);

        Aws::DynamoDB::Model::ExecuteStatementOutcome outcome =
dynamoClient.ExecuteStatement(
                request);
        if (!outcome.IsSuccess()) {
            std::cerr << "Failed to retrieve the movie information: "
                      << outcome.GetError().GetMessage() << std::endl;
            return false;
        }
```

```
        else {
            const Aws::DynamoDB::Model::ExecuteStatementResult &result =
outcome.GetResult();

            const Aws::Vector<Aws::Map<Aws::String,
Aws::DynamoDB::Model::AttributeValue>> &items = result.GetItems();

            if (items.size() == 1) {
                printMovieInfo(items[0]);
            }
            else {
                std::cerr << "Error: " << items.size() << " movies were
retrieved. "
                          << " There should be only one movie." << std::endl;
            }
        }
    }

    std::cout << "Deleting the movie" << std::endl;

    // 6. Delete the movie using a "Delete" statement. (ExecuteStatement)
    {
        Aws::DynamoDB::Model::ExecuteStatementRequest request;
        std::stringstream sqlStream;
        sqlStream << "DELETE FROM  \"" << MOVIE_TABLE_NAME << "\" WHERE "
                  << TITLE_KEY << "=? and " << YEAR_KEY << "=?";

        request.SetStatement(sqlStream.str());

        Aws::Vector<Aws::DynamoDB::Model::AttributeValue> attributes;
        attributes.push_back(Aws::DynamoDB::Model::AttributeValue().SetS(title));
        attributes.push_back(Aws::DynamoDB::Model::AttributeValue().SetN(year));
        request.SetParameters(attributes);

        Aws::DynamoDB::Model::ExecuteStatementOutcome outcome =
dynamoClient.ExecuteStatement(
                request);
        if (!outcome.IsSuccess()) {
            std::cerr << "Failed to delete the movie: "
                      << outcome.GetError().GetMessage() << std::endl;
            return false;
        }
    }
```

```
    std::cout << "Movie successfully deleted." << std::endl;
    return true;
}

//! Create a DynamoDB table to be used in sample code scenarios.
/*!
  \sa createMoviesDynamoDBTable()
  \param clientConfiguration: AWS client configuration.
  \return bool: Function succeeded.
*/
bool AwsDoc::DynamoDB::createMoviesDynamoDBTable(
        const Aws::Client::ClientConfiguration &clientConfiguration) {
    Aws::DynamoDB::DynamoDBClient dynamoClient(clientConfiguration);

    bool movieTableAlreadyExisted = false;

    {
        Aws::DynamoDB::Model::CreateTableRequest request;

        Aws::DynamoDB::Model::AttributeDefinition yearAttributeDefinition;
        yearAttributeDefinition.SetAttributeName(YEAR_KEY);
        yearAttributeDefinition.SetAttributeType(
                Aws::DynamoDB::Model::ScalarAttributeType::N);
        request.AddAttributeDefinitions(yearAttributeDefinition);

        Aws::DynamoDB::Model::AttributeDefinition titleAttributeDefinition;
        yearAttributeDefinition.SetAttributeName(TITLE_KEY);
        yearAttributeDefinition.SetAttributeType(
                Aws::DynamoDB::Model::ScalarAttributeType::S);
        request.AddAttributeDefinitions(yearAttributeDefinition);

        Aws::DynamoDB::Model::KeySchemaElement yearKeySchema;
        yearKeySchema.WithAttributeName(YEAR_KEY).WithKeyType(
                Aws::DynamoDB::Model::KeyType::HASH);
        request.AddKeySchema(yearKeySchema);

        Aws::DynamoDB::Model::KeySchemaElement titleKeySchema;
        yearKeySchema.WithAttributeName(TITLE_KEY).WithKeyType(
                Aws::DynamoDB::Model::KeyType::RANGE);
        request.AddKeySchema(yearKeySchema);

        Aws::DynamoDB::Model::ProvisionedThroughput throughput;
        throughput.WithReadCapacityUnits(
                PROVISIONED_THROUGHPUT_UNITS).WithWriteCapacityUnits(
```

```
                PROVISIONED_THROUGHPUT_UNITS);
        request.SetProvisionedThroughput(throughput);
        request.SetTableName(MOVIE_TABLE_NAME);

        std::cout << "Creating table '" << MOVIE_TABLE_NAME << "'..." <<
 std::endl;
        const Aws::DynamoDB::Model::CreateTableOutcome &result =
 dynamoClient.CreateTable(
                request);
        if (!result.IsSuccess()) {
            if (result.GetError().GetErrorType() ==
                Aws::DynamoDB::DynamoDBErrors::RESOURCE_IN_USE) {
                std::cout << "Table already exists." << std::endl;
                movieTableAlreadyExisted = true;
            }
            else {
                std::cerr << "Failed to create table: "
                          << result.GetError().GetMessage();
                return false;
            }
        }
    }

    // Wait for table to become active.
    if (!movieTableAlreadyExisted) {
        std::cout << "Waiting for table '" << MOVIE_TABLE_NAME
                  << "' to become active...." << std::endl;
        if (!AwsDoc::DynamoDB::waitTableActive(MOVIE_TABLE_NAME,
 clientConfiguration)) {
            return false;
        }
        std::cout << "Table '" << MOVIE_TABLE_NAME << "' created and active."
                  << std::endl;
    }

    return true;
}

//! Delete the DynamoDB table used for sample code scenarios.
/*!
  \sa deleteMoviesDynamoDBTable()
  \param clientConfiguration: AWS client configuration.
  \return bool: Function succeeded.
*/
```

```
bool AwsDoc::DynamoDB::deleteMoviesDynamoDBTable(
        const Aws::Client::ClientConfiguration &clientConfiguration) {
    Aws::DynamoDB::DynamoDBClient dynamoClient(clientConfiguration);

    Aws::DynamoDB::Model::DeleteTableRequest request;
    request.SetTableName(MOVIE_TABLE_NAME);

    const Aws::DynamoDB::Model::DeleteTableOutcome &result =
 dynamoClient.DeleteTable(
            request);
    if (result.IsSuccess()) {
        std::cout << "Your table \""
                  << result.GetResult().GetTableDescription().GetTableName()
                  << " was deleted.\n";
    }
    else {
        std::cerr << "Failed to delete table: " << result.GetError().GetMessage()
                  << std::endl;
    }

    return result.IsSuccess();
}

//! Query a newly created DynamoDB table until it is active.
/*!
  \sa waitTableActive()
  \param waitTableActive: The DynamoDB table's name.
  \param dynamoClient: A DynamoDB client.
  \return bool: Function succeeded.
*/
bool AwsDoc::DynamoDB::waitTableActive(const Aws::String &tableName,
                                       const Aws::DynamoDB::DynamoDBClient
 &dynamoClient) {

    // Repeatedly call DescribeTable until table is ACTIVE.
    const int MAX_QUERIES = 20;
    Aws::DynamoDB::Model::DescribeTableRequest request;
    request.SetTableName(tableName);

    int count = 0;
    while (count < MAX_QUERIES) {
        const Aws::DynamoDB::Model::DescribeTableOutcome &result =
 dynamoClient.DescribeTable(
                request);
```

```
        if (result.IsSuccess()) {
            Aws::DynamoDB::Model::TableStatus status =
 result.GetResult().GetTable().GetTableStatus();

            if (Aws::DynamoDB::Model::TableStatus::ACTIVE != status) {
                std::this_thread::sleep_for(std::chrono::seconds(1));
            }
            else {
                return true;
            }
        }
        else {
            std::cerr << "Error DynamoDB::waitTableActive "
                      << result.GetError().GetMessage() << std::endl;
            return false;
        }
        count++;
    }
    return false;
}
```

- For API details, see ExecuteStatement in *AWS SDK for C++ API Reference*.

Go

SDK for Go V2

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Run a scenario that creates a table and runs PartiQL queries.

```
import (
 "context"
 "fmt"
 "log"
 "strings"
```

```
 "time"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb"
 "github.com/awsdocs/aws-doc-sdk-examples/gov2/dynamodb/actions"
)

// RunPartiQLSingleScenario shows you how to use the AWS SDK for Go
// to use PartiQL to query a table that stores data about movies.
//
// * Use PartiQL statements to add, get, update, and delete data for individual
 movies.
//
// This example creates an Amazon DynamoDB service client from the specified
 sdkConfig so that
// you can replace it with a mocked or stubbed config for unit testing.
//
// This example creates and deletes a DynamoDB table to use during the scenario.
func RunPartiQLSingleScenario(ctx context.Context, sdkConfig aws.Config,
 tableName string) {
 defer func() {
  if r := recover(); r != nil {
   fmt.Printf("Something went wrong with the demo.")
  }
 }()

 log.Println(strings.Repeat("-", 88))
 log.Println("Welcome to the Amazon DynamoDB PartiQL single action demo.")
 log.Println(strings.Repeat("-", 88))

 tableBasics := actions.TableBasics{
  DynamoDbClient: dynamodb.NewFromConfig(sdkConfig),
  TableName:      tableName,
 }
 runner := actions.PartiQLRunner{
  DynamoDbClient: dynamodb.NewFromConfig(sdkConfig),
  TableName:      tableName,
 }

 exists, err := tableBasics.TableExists(ctx)
 if err != nil {
  panic(err)
 }
 if !exists {
```

```
 log.Printf("Creating table %v...\n", tableName)
 _, err = tableBasics.CreateMovieTable(ctx)
 if err != nil {
  panic(err)
 } else {
  log.Printf("Created table %v.\n", tableName)
 }
} else {
 log.Printf("Table %v already exists.\n", tableName)
}
log.Println(strings.Repeat("-", 88))

currentYear, _, _ := time.Now().Date()
customMovie := actions.Movie{
 Title: "24 Hour PartiQL People",
 Year:  currentYear,
 Info: map[string]interface{}{
  "plot":   "A group of data developers discover a new query language they can't
stop using.",
  "rating": 9.9,
 },
}

log.Printf("Inserting movie '%v' released in %v.", customMovie.Title,
customMovie.Year)
err = runner.AddMovie(ctx, customMovie)
if err == nil {
 log.Printf("Added %v to the movie table.\n", customMovie.Title)
}
log.Println(strings.Repeat("-", 88))

log.Printf("Getting data for movie '%v' released in %v.", customMovie.Title,
customMovie.Year)
movie, err := runner.GetMovie(ctx, customMovie.Title, customMovie.Year)
if err == nil {
 log.Println(movie)
}
log.Println(strings.Repeat("-", 88))

newRating := 6.6
log.Printf("Updating movie '%v' with a rating of %v.", customMovie.Title,
newRating)
err = runner.UpdateMovie(ctx, customMovie, newRating)
if err == nil {
```

```
  log.Printf("Updated %v with a new rating.\n", customMovie.Title)
 }
 log.Println(strings.Repeat("-", 88))

 log.Printf("Getting data again to verify the update.")
 movie, err = runner.GetMovie(ctx, customMovie.Title, customMovie.Year)
 if err == nil {
  log.Println(movie)
 }
 log.Println(strings.Repeat("-", 88))

 log.Printf("Deleting movie '%v'.\n", customMovie.Title)
 err = runner.DeleteMovie(ctx, customMovie)
 if err == nil {
  log.Printf("Deleted %v.\n", customMovie.Title)
 }

 err = tableBasics.DeleteTable(ctx)
 if err == nil {
  log.Printf("Deleted table %v.\n", tableBasics.TableName)
 }

 log.Println(strings.Repeat("-", 88))
 log.Println("Thanks for watching!")
 log.Println(strings.Repeat("-", 88))
}
```

Define a Movie struct that is used in this example.

```
import (
 "archive/zip"
 "bytes"
 "encoding/json"
 "fmt"
 "io"
 "log"
 "net/http"

 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/attributevalue"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb/types"
```

```
)

// Movie encapsulates data about a movie. Title and Year are the composite
 primary key
// of the movie in Amazon DynamoDB. Title is the sort key, Year is the partition
 key,
// and Info is additional data.
type Movie struct {
 Title string                 `dynamodbav:"title"`
 Year  int                    `dynamodbav:"year"`
 Info  map[string]interface{} `dynamodbav:"info"`
}

// GetKey returns the composite primary key of the movie in a format that can be
// sent to DynamoDB.
func (movie Movie) GetKey() map[string]types.AttributeValue {
 title, err := attributevalue.Marshal(movie.Title)
 if err != nil {
  panic(err)
 }
 year, err := attributevalue.Marshal(movie.Year)
 if err != nil {
  panic(err)
 }
 return map[string]types.AttributeValue{"title": title, "year": year}
}

// String returns the title, year, rating, and plot of a movie, formatted for the
 example.
func (movie Movie) String() string {
 return fmt.Sprintf("%v\n\tReleased: %v\n\tRating: %v\n\tPlot: %v\n",
  movie.Title, movie.Year, movie.Info["rating"], movie.Info["plot"])
}
```

Create a struct and methods that run PartiQL statements.

```
import (
 "context"
 "fmt"
 "log"
```

```
 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/attributevalue"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb/types"
)

// PartiQLRunner encapsulates the Amazon DynamoDB service actions used in the
// PartiQL examples. It contains a DynamoDB service client that is used to act on
 the
// specified table.
type PartiQLRunner struct {
 DynamoDbClient *dynamodb.Client
 TableName      string
}



// AddMovie runs a PartiQL INSERT statement to add a movie to the DynamoDB table.
func (runner PartiQLRunner) AddMovie(ctx context.Context, movie Movie) error {
 params, err := attributevalue.MarshalList([]interface{}{movie.Title, movie.Year,
 movie.Info})
 if err != nil {
  panic(err)
 }
 _, err = runner.DynamoDbClient.ExecuteStatement(ctx,
 &dynamodb.ExecuteStatementInput{
  Statement: aws.String(
   fmt.Sprintf("INSERT INTO \"%v\" VALUE {'title': ?, 'year': ?, 'info': ?}",
    runner.TableName)),
  Parameters: params,
 })
 if err != nil {
  log.Printf("Couldn't insert an item with PartiQL. Here's why: %v\n", err)
 }
 return err
}



// GetMovie runs a PartiQL SELECT statement to get a movie from the DynamoDB
 table by
// title and year.
```

```
func (runner PartiQLRunner) GetMovie(ctx context.Context, title string, year int)
 (Movie, error) {
 var movie Movie
 params, err := attributevalue.MarshalList([]interface{}{title, year})
 if err != nil {
  panic(err)
 }
 response, err := runner.DynamoDbClient.ExecuteStatement(ctx,
 &dynamodb.ExecuteStatementInput{
  Statement: aws.String(
   fmt.Sprintf("SELECT * FROM \"%v\" WHERE title=? AND year=?",
    runner.TableName)),
  Parameters: params,
 })
 if err != nil {
  log.Printf("Couldn't get info about %v. Here's why: %v\n", title, err)
 } else {
  err = attributevalue.UnmarshalMap(response.Items[0], &movie)
  if err != nil {
   log.Printf("Couldn't unmarshal response. Here's why: %v\n", err)
  }
 }
 return movie, err
}



// UpdateMovie runs a PartiQL UPDATE statement to update the rating of a movie
 that
// already exists in the DynamoDB table.
func (runner PartiQLRunner) UpdateMovie(ctx context.Context, movie Movie, rating
 float64) error {
 params, err := attributevalue.MarshalList([]interface{}{rating, movie.Title,
 movie.Year})
 if err != nil {
  panic(err)
 }
 _, err = runner.DynamoDbClient.ExecuteStatement(ctx,
 &dynamodb.ExecuteStatementInput{
  Statement: aws.String(
   fmt.Sprintf("UPDATE \"%v\" SET info.rating=? WHERE title=? AND year=?",
    runner.TableName)),
  Parameters: params,
 })
```

```
 if err != nil {
  log.Printf("Couldn't update movie %v. Here's why: %v\n", movie.Title, err)
 }
 return err
}



// DeleteMovie runs a PartiQL DELETE statement to remove a movie from the
 DynamoDB table.
func (runner PartiQLRunner) DeleteMovie(ctx context.Context, movie Movie) error {
 params, err := attributevalue.MarshalList([]interface{}{movie.Title,
 movie.Year})
 if err != nil {
  panic(err)
 }
 _, err = runner.DynamoDbClient.ExecuteStatement(ctx,
 &dynamodb.ExecuteStatementInput{
  Statement: aws.String(
   fmt.Sprintf("DELETE FROM \"%v\" WHERE title=? AND year=?",
    runner.TableName)),
  Parameters: params,
 })
 if err != nil {
  log.Printf("Couldn't delete %v from the table. Here's why: %v\n", movie.Title,
 err)
 }
 return err
}
```

- For API details, see ExecuteStatement in *AWS SDK for Go API Reference*.

Java

SDK for Java 2.x

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
public class ScenarioPartiQ {
    public static void main(String[] args) throws IOException {
        String fileName = "../../../resources/sample_files/movies.json";
        String tableName = "MoviesPartiQ";
        Region region = Region.US_EAST_1;
        DynamoDbClient ddb = DynamoDbClient.builder()
            .region(region)
            .build();

        System.out.println(
            "******* Creating an Amazon DynamoDB table named MoviesPartiQ with a
 key named year and a sort key named title.");
        createTable(ddb, tableName);

        System.out.println("Loading data into the MoviesPartiQ table.");
        loadData(ddb, fileName);

        System.out.println("Getting data from the MoviesPartiQ table.");
        getItem(ddb);

        System.out.println("Putting a record into the MoviesPartiQ table.");
        putRecord(ddb);

        System.out.println("Updating a record.");
        updateTableItem(ddb);

        System.out.println("Querying the movies released in 2013.");
        queryTable(ddb);

        System.out.println("Deleting the Amazon DynamoDB table.");
        deleteDynamoDBTable(ddb, tableName);
        ddb.close();
    }

    public static void createTable(DynamoDbClient ddb, String tableName) {
        DynamoDbWaiter dbWaiter = ddb.waiter();
        ArrayList<AttributeDefinition> attributeDefinitions = new ArrayList<>();

        // Define attributes.
        attributeDefinitions.add(AttributeDefinition.builder()
            .attributeName("year")
            .attributeType("N")
            .build());
```

```
        attributeDefinitions.add(AttributeDefinition.builder()
            .attributeName("title")
            .attributeType("S")
            .build());

        ArrayList<KeySchemaElement> tableKey = new ArrayList<>();
        KeySchemaElement key = KeySchemaElement.builder()
            .attributeName("year")
            .keyType(KeyType.HASH)
            .build();

        KeySchemaElement key2 = KeySchemaElement.builder()
            .attributeName("title")
            .keyType(KeyType.RANGE) // Sort
            .build();

        // Add KeySchemaElement objects to the list.
        tableKey.add(key);
        tableKey.add(key2);

        CreateTableRequest request = CreateTableRequest.builder()
            .keySchema(tableKey)
            .billingMode(BillingMode.PAY_PER_REQUEST) //Scales based on traffic.
            .attributeDefinitions(attributeDefinitions)
            .tableName(tableName)
            .build();

        try {
            CreateTableResponse response = ddb.createTable(request);
            DescribeTableRequest tableRequest = DescribeTableRequest.builder()
                .tableName(tableName)
                .build();

            // Wait until the Amazon DynamoDB table is created.
            WaiterResponse<DescribeTableResponse> waiterResponse =
dbWaiter.waitUntilTableExists(tableRequest);
            waiterResponse.matched().response().ifPresent(System.out::println);
            String newTable = response.tableDescription().tableName();
            System.out.println("The " + newTable + " was successfully created.");

        } catch (DynamoDbException e) {
            System.err.println(e.getMessage());
            System.exit(1);
```

```
        }
    }

    // Load data into the table.
    public static void loadData(DynamoDbClient ddb, String fileName) throws
IOException {

        String sqlStatement = "INSERT INTO MoviesPartiQ VALUE {'year':?,
'title' : ?, 'info' : ?}";
        JsonParser parser = new JsonFactory().createParser(new File(fileName));
        com.fasterxml.jackson.databind.JsonNode rootNode = new
ObjectMapper().readTree(parser);
        Iterator<JsonNode> iter = rootNode.iterator();
        ObjectNode currentNode;
        int t = 0;
        List<AttributeValue> parameters = new ArrayList<>();
        while (iter.hasNext()) {

            // Add 200 movies to the table.
            if (t == 200)
                break;
            currentNode = (ObjectNode) iter.next();

            int year = currentNode.path("year").asInt();
            String title = currentNode.path("title").asText();
            String info = currentNode.path("info").toString();

            AttributeValue att1 = AttributeValue.builder()
                .n(String.valueOf(year))
                .build();

            AttributeValue att2 = AttributeValue.builder()
                .s(title)
                .build();

            AttributeValue att3 = AttributeValue.builder()
                .s(info)
                .build();

            parameters.add(att1);
            parameters.add(att2);
            parameters.add(att3);

            // Insert the movie into the Amazon DynamoDB table.
```

```
            executeStatementRequest(ddb, sqlStatement, parameters);
            System.out.println("Added Movie " + title);

            parameters.remove(att1);
            parameters.remove(att2);
            parameters.remove(att3);
            t++;
        }
    }

    public static void getItem(DynamoDbClient ddb) {

        String sqlStatement = "SELECT * FROM MoviesPartiQ where year=? and
title=?";
        List<AttributeValue> parameters = new ArrayList<>();
        AttributeValue att1 = AttributeValue.builder()
            .n("2012")
            .build();

        AttributeValue att2 = AttributeValue.builder()
            .s("The Perks of Being a Wallflower")
            .build();

        parameters.add(att1);
        parameters.add(att2);

        try {
            ExecuteStatementResponse response = executeStatementRequest(ddb,
sqlStatement, parameters);
            System.out.println("ExecuteStatement successful: " +
response.toString());

        } catch (DynamoDbException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }

    public static void putRecord(DynamoDbClient ddb) {

        String sqlStatement = "INSERT INTO MoviesPartiQ VALUE {'year':?,
'title' : ?, 'info' : ?}";
        try {
            List<AttributeValue> parameters = new ArrayList<>();
```

```
            AttributeValue att1 = AttributeValue.builder()
                .n(String.valueOf("2020"))
                .build();

            AttributeValue att2 = AttributeValue.builder()
                .s("My Movie")
                .build();

            AttributeValue att3 = AttributeValue.builder()
                .s("No Information")
                .build();

            parameters.add(att1);
            parameters.add(att2);
            parameters.add(att3);

            executeStatementRequest(ddb, sqlStatement, parameters);
            System.out.println("Added new movie.");

        } catch (DynamoDbException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }

    public static void updateTableItem(DynamoDbClient ddb) {

        String sqlStatement = "UPDATE MoviesPartiQ SET info = 'directors\":
[\"Merian C. Cooper\",\"Ernest B. Schoedsack' where year=? and title=?";
        List<AttributeValue> parameters = new ArrayList<>();
        AttributeValue att1 = AttributeValue.builder()
            .n(String.valueOf("2013"))
            .build();

        AttributeValue att2 = AttributeValue.builder()
            .s("The East")
            .build();

        parameters.add(att1);
        parameters.add(att2);

        try {
            executeStatementRequest(ddb, sqlStatement, parameters);
```

```
        } catch (DynamoDbException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
        System.out.println("Item was updated!");
    }

    // Query the table where the year is 2013.
    public static void queryTable(DynamoDbClient ddb) {
        String sqlStatement = "SELECT * FROM MoviesPartiQ where year = ? ORDER BY
year";
        try {

            List<AttributeValue> parameters = new ArrayList<>();
            AttributeValue att1 = AttributeValue.builder()
                .n(String.valueOf("2013"))
                .build();
            parameters.add(att1);

            // Get items in the table and write out the ID value.
            ExecuteStatementResponse response = executeStatementRequest(ddb,
sqlStatement, parameters);
            System.out.println("ExecuteStatement successful: " +
response.toString());

        } catch (DynamoDbException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }

    public static void deleteDynamoDBTable(DynamoDbClient ddb, String tableName)
{

        DeleteTableRequest request = DeleteTableRequest.builder()
            .tableName(tableName)
            .build();

        try {
            ddb.deleteTable(request);

        } catch (DynamoDbException e) {
            System.err.println(e.getMessage());
```

```
            System.exit(1);
        }
        System.out.println(tableName + " was successfully deleted!");
    }

    private static ExecuteStatementResponse
 executeStatementRequest(DynamoDbClient ddb, String statement,

 List<AttributeValue> parameters) {
        ExecuteStatementRequest request = ExecuteStatementRequest.builder()
            .statement(statement)
            .parameters(parameters)
            .build();

        return ddb.executeStatement(request);
    }

    private static void processResults(ExecuteStatementResponse
 executeStatementResult) {
        System.out.println("ExecuteStatement successful: " +
 executeStatementResult.toString());
    }
}
```

- For API details, see ExecuteStatement in *AWS SDK for Java 2.x API Reference*.

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Execute single PartiQL statements.

```
import {
  BillingMode,
  CreateTableCommand,
```

```
  DeleteTableCommand,
  DescribeTableCommand,
  DynamoDBClient,
  waitUntilTableExists,
} from "@aws-sdk/client-dynamodb";
import {
  DynamoDBDocumentClient,
  ExecuteStatementCommand,
} from "@aws-sdk/lib-dynamodb";
import { ScenarioInput } from "@aws-doc-sdk-examples/lib/scenario";

const client = new DynamoDBClient({});
const docClient = DynamoDBDocumentClient.from(client);

const log = (msg) => console.log(`[SCENARIO] ${msg}`);
const tableName = "SingleOriginCoffees";

export const main = async (confirmAll = false) => {
  /**
   * Delete table if it exists.
   */
  try {
    await client.send(new DescribeTableCommand({ TableName: tableName }));
    // If no error was thrown, the table exists.
    const input = new ScenarioInput(
      "deleteTable",
      `A table named ${tableName} already exists. If you choose not to delete
this table, the scenario cannot continue. Delete it?`,
      { type: "confirm", confirmAll },
    );
    const deleteTable = await input.handle({});
    if (deleteTable) {
      await client.send(new DeleteTableCommand({ tableName }));
    } else {
      console.warn(
        "Scenario could not run. Either delete ${tableName} or provide a unique
 table name.",
      );
      return;
    }
  } catch (caught) {
    if (
      caught instanceof Error &&
      caught.name === "ResourceNotFoundException"
```

```
    ) {
      // Do nothing. This means the table is not there.
    } else {
      throw caught;
    }
  }

  /**
   * Create a table.
   */

  log("Creating a table.");
  const createTableCommand = new CreateTableCommand({
    TableName: tableName,
    // This example performs a large write to the database.
    // Set the billing mode to PAY_PER_REQUEST to
    // avoid throttling the large write.
    BillingMode: BillingMode.PAY_PER_REQUEST,
    // Define the attributes that are necessary for the key schema.
    AttributeDefinitions: [
      {
        AttributeName: "varietal",
        // 'S' is a data type descriptor that represents a number type.
        // For a list of all data type descriptors, see the following link.
        // https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/
Programming.LowLevelAPI.html#Programming.LowLevelAPI.DataTypeDescriptors
        AttributeType: "S",
      },
    ],
    // The KeySchema defines the primary key. The primary key can be
    // a partition key, or a combination of a partition key and a sort key.
    // Key schema design is important. For more info, see
    // https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/best-
practices.html
    KeySchema: [{ AttributeName: "varietal", KeyType: "HASH" }],
  });
  await client.send(createTableCommand);
  log(`Table created: ${tableName}.`);

  /**
   * Wait until the table is active.
   */

  // This polls with DescribeTableCommand until the requested table is 'ACTIVE'.
```

```
  // You can't write to a table before it's active.
  log("Waiting for the table to be active.");
  await waitUntilTableExists({ client }, { TableName: tableName });
  log("Table active.");

  /**
   * Insert an item.
   */

  log("Inserting a coffee into the table.");
  const addItemStatementCommand = new ExecuteStatementCommand({
    // https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/ql-
reference.insert.html
    Statement: `INSERT INTO ${tableName} value {'varietal':?, 'profile':?}`,
    Parameters: ["arabica", ["chocolate", "floral"]],
  });
  await client.send(addItemStatementCommand);
  log("Coffee inserted.");

  /**
   * Select an item.
   */

  log("Selecting the coffee from the table.");
  const selectItemStatementCommand = new ExecuteStatementCommand({
    // https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/ql-
reference.select.html
    Statement: `SELECT * FROM ${tableName} WHERE varietal=?`,
    Parameters: ["arabica"],
  });
  const selectItemResponse = await docClient.send(selectItemStatementCommand);
  log(`Got coffee: ${JSON.stringify(selectItemResponse.Items[0])}`);

  /**
   * Update the item.
   */

  log("Add a flavor profile to the coffee.");
  const updateItemStatementCommand = new ExecuteStatementCommand({
    // https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/ql-
reference.update.html
    Statement: `UPDATE ${tableName} SET profile=list_append(profile, ?) WHERE
 varietal=?`,
    Parameters: [["fruity"], "arabica"],
```

```
  });
  await client.send(updateItemStatementCommand);
  log("Updated coffee");

  /**
   * Delete the item.
   */

  log("Deleting the coffee.");
  const deleteItemStatementCommand = new ExecuteStatementCommand({
    // https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/ql-
reference.delete.html
    Statement: `DELETE FROM ${tableName} WHERE varietal=?`,
    Parameters: ["arabica"],
  });
  await docClient.send(deleteItemStatementCommand);
  log("Coffee deleted.");

  /**
   * Delete the table.
   */

  log("Deleting the table.");
  const deleteTableCommand = new DeleteTableCommand({ TableName: tableName });
  await client.send(deleteTableCommand);
  log("Table deleted.");
};
```

- For API details, see ExecuteStatement in *AWS SDK for JavaScript API Reference*.

Kotlin

SDK for Kotlin

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
suspend fun main() {
```

```
    val ddb = DynamoDbClient.fromEnvironment { region = "us-east-1" }
    val tableName = "MoviesPartiQ"
    val fileName = "../../../resources/sample_files/movies.json"
    println("Creating an Amazon DynamoDB table named MoviesPartiQ with a key
 named id and a sort key named title.")
    createTablePartiQL(ddb, tableName, "year")
    loadDataPartiQL(ddb, fileName)

    println("******* Getting data from the MoviesPartiQ table.")
    getMoviePartiQL(ddb)

    println("******* Putting a record into the MoviesPartiQ table.")
    putRecordPartiQL(ddb)

    println("******* Updating a record.")
    updateTableItemPartiQL(ddb)

    println("******* Querying the movies released in 2013.")
    queryTablePartiQL(ddb)

    println("******* Deleting the MoviesPartiQ table.")
    deleteTablePartiQL(tableName)
}

suspend fun createTablePartiQL(
    ddb: DynamoDbClient,
    tableNameVal: String,
    key: String,
) {
    val attDef =
        AttributeDefinition {
            attributeName = key
            attributeType = ScalarAttributeType.N
        }

    val attDef1 =
        AttributeDefinition {
            attributeName = "title"
            attributeType = ScalarAttributeType.S
        }

    val keySchemaVal =
        KeySchemaElement {
            attributeName = key
```

```
            keyType = KeyType.Hash
        }

    val keySchemaVal1 =
        KeySchemaElement {
            attributeName = "title"
            keyType = KeyType.Range
        }

    val request =
        CreateTableRequest {
            attributeDefinitions = listOf(attDef, attDef1)
            keySchema = listOf(keySchemaVal, keySchemaVal1)
            billingMode = BillingMode.PayPerRequest
            tableName = tableNameVal
        }

    val response = ddb.createTable(request)
    ddb.waitUntilTableExists {
        // suspend call
        tableName = tableNameVal
    }
    println("The table was successfully created
 ${response.tableDescription?.tableArn}")
}

suspend fun loadDataPartiQL(
    ddb: DynamoDbClient,
    fileName: String,
) {
    val sqlStatement = "INSERT INTO MoviesPartiQ VALUE {'year':?, 'title' : ?,
 'info' : ?}"
    val parser = JsonFactory().createParser(File(fileName))
    val rootNode = ObjectMapper().readTree<JsonNode>(parser)
    val iter: Iterator<JsonNode> = rootNode.iterator()
    var currentNode: ObjectNode
    var t = 0

    while (iter.hasNext()) {
        if (t == 200) {
            break
        }

        currentNode = iter.next() as ObjectNode
```

```
        val year = currentNode.path("year").asInt()
        val title = currentNode.path("title").asText()
        val info = currentNode.path("info").toString()

        val parameters: MutableList<AttributeValue> = ArrayList<AttributeValue>()
        parameters.add(AttributeValue.N(year.toString()))
        parameters.add(AttributeValue.S(title))
        parameters.add(AttributeValue.S(info))

        executeStatementPartiQL(ddb, sqlStatement, parameters)
        println("Added Movie $title")
        parameters.clear()
        t++
    }
}

suspend fun getMoviePartiQL(ddb: DynamoDbClient) {
    val sqlStatement = "SELECT * FROM MoviesPartiQ where year=? and title=?"
    val parameters: MutableList<AttributeValue> = ArrayList<AttributeValue>()
    parameters.add(AttributeValue.N("2012"))
    parameters.add(AttributeValue.S("The Perks of Being a Wallflower"))
    val response = executeStatementPartiQL(ddb, sqlStatement, parameters)
    println("ExecuteStatement successful: $response")
}

suspend fun putRecordPartiQL(ddb: DynamoDbClient) {
    val sqlStatement = "INSERT INTO MoviesPartiQ VALUE {'year':?, 'title' : ?,
 'info' : ?}"
    val parameters: MutableList<AttributeValue> = java.util.ArrayList()
    parameters.add(AttributeValue.N("2020"))
    parameters.add(AttributeValue.S("My Movie"))
    parameters.add(AttributeValue.S("No Info"))
    executeStatementPartiQL(ddb, sqlStatement, parameters)
    println("Added new movie.")
}

suspend fun updateTableItemPartiQL(ddb: DynamoDbClient) {
    val sqlStatement = "UPDATE MoviesPartiQ SET info = 'directors\":[\"Merian C.
 Cooper\",\"Ernest B. Schoedsack\' where year=? and title=?"
    val parameters: MutableList<AttributeValue> = java.util.ArrayList()
    parameters.add(AttributeValue.N("2013"))
    parameters.add(AttributeValue.S("The East"))
    executeStatementPartiQL(ddb, sqlStatement, parameters)
    println("Item was updated!")
```

```
}

// Query the table where the year is 2013.
suspend fun queryTablePartiQL(ddb: DynamoDbClient) {
    val sqlStatement = "SELECT * FROM MoviesPartiQ where year = ?"
    val parameters: MutableList<AttributeValue> = java.util.ArrayList()
    parameters.add(AttributeValue.N("2013"))
    val response = executeStatementPartiQL(ddb, sqlStatement, parameters)
    println("ExecuteStatement successful: $response")
}

suspend fun deleteTablePartiQL(tableNameVal: String) {
    val request =
        DeleteTableRequest {
            tableName = tableNameVal
        }

    DynamoDbClient { region = "us-east-1" }.use { ddb ->
        ddb.deleteTable(request)
        println("$tableNameVal was deleted")
    }
}

suspend fun executeStatementPartiQL(
    ddb: DynamoDbClient,
    statementVal: String,
    parametersVal: List<AttributeValue>,
): ExecuteStatementResponse {
    val request =
        ExecuteStatementRequest {
            statement = statementVal
            parameters = parametersVal
        }

    return ddb.executeStatement(request)
}
```

- For API details, see ExecuteStatement in *AWS SDK for Kotlin API reference*.

PHP

SDK for PHP

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
namespace DynamoDb\PartiQL_Basics;

use Aws\DynamoDb\Marshaler;
use DynamoDb;
use DynamoDb\DynamoDBAttribute;

use function AwsUtilities\testable_readline;
use function AwsUtilities\loadMovieData;

class GettingStartedWithPartiQL
{
    public function run()
    {
        echo("\n");
        echo("--------------------------------------\n");
        print("Welcome to the Amazon DynamoDB - PartiQL getting started demo
 using PHP!\n");
        echo("--------------------------------------\n");

        $uuid = uniqid();
        $service = new DynamoDb\DynamoDBService();

        $tableName = "partiql_demo_table_$uuid";
        $service->createTable(
            $tableName,
            [
                new DynamoDBAttribute('year', 'N', 'HASH'),
                new DynamoDBAttribute('title', 'S', 'RANGE')
            ]
        );

        echo "Waiting for table...";
```

```
        $service->dynamoDbClient->waitUntil("TableExists", ['TableName' =>
 $tableName]);
        echo "table $tableName found!\n";

        echo "What's the name of the last movie you watched?\n";
        while (empty($movieName)) {
            $movieName = testable_readline("Movie name: ");
        }
        echo "And what year was it released?\n";
        $movieYear = "year";
        while (!is_numeric($movieYear) || intval($movieYear) != $movieYear) {
            $movieYear = testable_readline("Year released: ");
        }
        $key = [
            'Item' => [
                'year' => [
                    'N' => "$movieYear",
                ],
                'title' => [
                    'S' => $movieName,
                ],
            ],
        ];
        list($statement, $parameters) = $service-
>buildStatementAndParameters("INSERT", $tableName, $key);
        $service->insertItemByPartiQL($statement, $parameters);

        echo "How would you rate the movie from 1-10?\n";
        $rating = 0;
        while (!is_numeric($rating) || intval($rating) != $rating || $rating < 1
 || $rating > 10) {
            $rating = testable_readline("Rating (1-10): ");
        }
        echo "What was the movie about?\n";
        while (empty($plot)) {
            $plot = testable_readline("Plot summary: ");
        }
        $attributes = [
            new DynamoDBAttribute('rating', 'N', 'HASH', $rating),
            new DynamoDBAttribute('plot', 'S', 'RANGE', $plot),
        ];

        list($statement, $parameters) = $service-
>buildStatementAndParameters("UPDATE", $tableName, $key, $attributes);
```

```
        $service->updateItemByPartiQL($statement, $parameters);
        echo "Movie added and updated.\n";



        $batch = json_decode(loadMovieData());

        $service->writeBatch($tableName, $batch);

        $movie = $service->getItemByPartiQL($tableName, $key);
        echo "\nThe movie {$movie['Items'][0]['title']['S']} was released in
 {$movie['Items'][0]['year']['N']}.\n";
        echo "What rating would you like to give {$movie['Items'][0]['title']
['S']}?\n";
        $rating = 0;
        while (!is_numeric($rating) || intval($rating) != $rating || $rating < 1
 || $rating > 10) {
            $rating = testable_readline("Rating (1-10): ");
        }
        $attributes = [
            new DynamoDBAttribute('rating', 'N', 'HASH', $rating),
            new DynamoDBAttribute('plot', 'S', 'RANGE', $plot)
        ];
        list($statement, $parameters) = $service-
>buildStatementAndParameters("UPDATE", $tableName, $key, $attributes);
        $service->updateItemByPartiQL($statement, $parameters);

        $movie = $service->getItemByPartiQL($tableName, $key);
        echo "Okay, you have rated {$movie['Items'][0]['title']['S']} as a
 {$movie['Items'][0]['rating']['N']}\n";

        $service->deleteItemByPartiQL($statement, $parameters);
        echo "But, bad news, this was a trap. That movie has now been deleted
 because of your rating...harsh.\n";

        echo "That's okay though. The book was better. Now, for something
 lighter, in what year were you born?\n";
        $birthYear = "not a number";
        while (!is_numeric($birthYear) || $birthYear >= date("Y")) {
            $birthYear = testable_readline("Birth year: ");
        }
        $birthKey = [
            'Key' => [
                'year' => [
```

```
                    'N' => "$birthYear",
                ],
            ],
        ];
        $result = $service->query($tableName, $birthKey);
        $marshal = new Marshaler();
        echo "Here are the movies in our collection released the year you were
 born:\n";
        $oops = "Oops! There were no movies released in that year (that we know
 of).\n";
        $display = "";
        foreach ($result['Items'] as $movie) {
            $movie = $marshal->unmarshalItem($movie);
            $display .= $movie['title'] . "\n";
        }
        echo ($display) ?: $oops;

        $yearsKey = [
            'Key' => [
                'year' => [
                    'N' => [
                        'minRange' => 1990,
                        'maxRange' => 1999,
                    ],
                ],
            ],
        ];
        $filter = "year between 1990 and 1999";
        echo "\nHere's a list of all the movies released in the 90s:\n";
        $result = $service->scan($tableName, $yearsKey, $filter);
        foreach ($result['Items'] as $movie) {
            $movie = $marshal->unmarshalItem($movie);
            echo $movie['title'] . "\n";
        }

        echo "\nCleaning up this demo by deleting table $tableName...\n";
        $service->deleteTable($tableName);
    }
}

    public function insertItemByPartiQL(string $statement, array $parameters)
    {
        $this->dynamoDbClient->executeStatement([
            'Statement' => "$statement",
```

```
            'Parameters' => $parameters,
        ]);
    }

    public function getItemByPartiQL(string $tableName, array $key): Result
    {
        list($statement, $parameters) = $this-
>buildStatementAndParameters("SELECT", $tableName, $key['Item']);

        return $this->dynamoDbClient->executeStatement([
            'Parameters' => $parameters,
            'Statement' => $statement,
        ]);
    }

    public function updateItemByPartiQL(string $statement, array $parameters)
    {
        $this->dynamoDbClient->executeStatement([
            'Statement' => $statement,
            'Parameters' => $parameters,
        ]);
    }

    public function deleteItemByPartiQL(string $statement, array $parameters)
    {
        $this->dynamoDbClient->executeStatement([
            'Statement' => $statement,
            'Parameters' => $parameters,
        ]);
    }
```

- For API details, see ExecuteStatement in *AWS SDK for PHP API Reference*.

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Create a class that can run PartiQL statements.

```
from datetime import datetime
from decimal import Decimal
import logging
from pprint import pprint

import boto3
from botocore.exceptions import ClientError

from scaffold import Scaffold

logger = logging.getLogger(__name__)

class PartiQLWrapper:
    """
    Encapsulates a DynamoDB resource to run PartiQL statements.
    """

    def __init__(self, dyn_resource):
        """
        :param dyn_resource: A Boto3 DynamoDB resource.
        """
        self.dyn_resource = dyn_resource


    def run_partiql(self, statement, params):
        """
        Runs a PartiQL statement. A Boto3 resource is used even though
        `execute_statement` is called on the underlying `client` object because
 the
        resource transforms input and output from plain old Python objects
 (POPOs) to
        the DynamoDB format. If you create the client directly, you must do these
        transforms yourself.

        :param statement: The PartiQL statement.
        :param params: The list of PartiQL parameters. These are applied to the
                       statement in the order they are listed.
        :return: The items returned from the statement, if any.
        """
        try:
            output = self.dyn_resource.meta.client.execute_statement(
                Statement=statement, Parameters=params
```

```
            )
        except ClientError as err:
            if err.response["Error"]["Code"] == "ResourceNotFoundException":
                logger.error(
                    "Couldn't execute PartiQL '%s' because the table does not
 exist.",
                    statement,
                )
            else:
                logger.error(
                    "Couldn't execute PartiQL '%s'. Here's why: %s: %s",
                    statement,
                    err.response["Error"]["Code"],
                    err.response["Error"]["Message"],
                )
            raise
        else:
            return output
```

Run a scenario that creates a table and runs PartiQL queries.

```
def run_scenario(scaffold, wrapper, table_name):
    logging.basicConfig(level=logging.INFO, format="%(levelname)s: %(message)s")

    print("-" * 88)
    print("Welcome to the Amazon DynamoDB PartiQL single statement demo.")
    print("-" * 88)

    print(f"Creating table '{table_name}' for the demo...")
    scaffold.create_table(table_name)
    print("-" * 88)

    title = "24 Hour PartiQL People"
    year = datetime.now().year
    plot = "A group of data developers discover a new query language they can't
 stop using."
    rating = Decimal("9.9")

    print(f"Inserting movie '{title}' released in {year}.")
    wrapper.run_partiql(
```

```
        f"INSERT INTO \"{table_name}\" VALUE {{'title': ?, 'year': ?,
 'info': ?}}",
        [title, year, {"plot": plot, "rating": rating}],
    )
    print("Success!")
    print("-" * 88)

    print(f"Getting data for movie '{title}' released in {year}.")
    output = wrapper.run_partiql(
        f'SELECT * FROM "{table_name}" WHERE title=? AND year=?', [title, year]
    )
    for item in output["Items"]:
        print(f"\n{item['title']}, {item['year']}")
        pprint(output["Items"])
    print("-" * 88)

    rating = Decimal("2.4")
    print(f"Updating movie '{title}' with a rating of {float(rating)}.")
    wrapper.run_partiql(
        f'UPDATE "{table_name}" SET info.rating=? WHERE title=? AND year=?',
        [rating, title, year],
    )
    print("Success!")
    print("-" * 88)

    print(f"Getting data again to verify our update.")
    output = wrapper.run_partiql(
        f'SELECT * FROM "{table_name}" WHERE title=? AND year=?', [title, year]
    )
    for item in output["Items"]:
        print(f"\n{item['title']}, {item['year']}")
        pprint(output["Items"])
    print("-" * 88)

    print(f"Deleting movie '{title}' released in {year}.")
    wrapper.run_partiql(
        f'DELETE FROM "{table_name}" WHERE title=? AND year=?', [title, year]
    )
    print("Success!")
    print("-" * 88)

    print(f"Deleting table '{table_name}'...")
    scaffold.delete_table()
    print("-" * 88)
```

```
    print("\nThanks for watching!")
    print("-" * 88)


if __name__ == "__main__":
    try:
        dyn_res = boto3.resource("dynamodb")
        scaffold = Scaffold(dyn_res)
        movies = PartiQLWrapper(dyn_res)
        run_scenario(scaffold, movies, "doc-example-table-partiql-movies")
    except Exception as e:
        print(f"Something went wrong with the demo! Here's what: {e}")
```

- For API details, see ExecuteStatement in *AWS SDK for Python (Boto3) API Reference*.

Ruby

SDK for Ruby

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Run a scenario that creates a table and runs PartiQL queries.

```
  table_name = "doc-example-table-movies-partiql-#{rand(10**8)}"
  scaffold = Scaffold.new(table_name)
  sdk = DynamoDBPartiQLSingle.new(table_name)

  new_step(1, 'Create a new DynamoDB table if none already exists.')
  unless scaffold.exists?(table_name)
    puts("\nNo such table: #{table_name}. Creating it...")
    scaffold.create_table(table_name)
    print "Done!\n".green
  end

  new_step(2, 'Populate DynamoDB table with movie data.')
  download_file = 'moviedata.json'
```

```
  puts("Downloading movie database to #{download_file}...")
  movie_data = scaffold.fetch_movie_data(download_file)
  puts("Writing movie data from #{download_file} into your table...")
  scaffold.write_batch(movie_data)
  puts("Records added: #{movie_data.length}.")
  print "Done!\n".green

  new_step(3, 'Select a single item from the movies table.')
  response = sdk.select_item_by_title('Star Wars')
  puts("Items selected for title 'Star Wars': #{response.items.length}\n")
  print response.items.first.to_s.yellow
  print "\n\nDone!\n".green

  new_step(4, 'Update a single item from the movies table.')
  puts "Let's correct the rating on The Big Lebowski to 10.0."
  sdk.update_rating_by_title('The Big Lebowski', 1998, 10.0)
  print "\nDone!\n".green

  new_step(5, 'Delete a single item from the movies table.')
  puts "Let's delete The Silence of the Lambs because it's just too scary."
  sdk.delete_item_by_title('The Silence of the Lambs', 1991)
  print "\nDone!\n".green

  new_step(6, 'Insert a new item into the movies table.')
  puts "Let's create a less-scary movie called The Prancing of the Lambs."
  sdk.insert_item('The Prancing of the Lambs', 2005, 'A movie about happy
 livestock.', 5.0)
  print "\nDone!\n".green

  new_step(7, 'Delete the table.')
  return unless scaffold.exists?(table_name)

  scaffold.delete_table
end
```

- For API details, see ExecuteStatement in *AWS SDK for Ruby API Reference*.

Rust

SDK for Rust

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
async fn make_table(
    client: &Client,
    table: &str,
    key: &str,
) -> Result<(), SdkError<CreateTableError>> {
    let ad = AttributeDefinition::builder()
        .attribute_name(key)
        .attribute_type(ScalarAttributeType::S)
        .build()
        .expect("creating AttributeDefinition");

    let ks = KeySchemaElement::builder()
        .attribute_name(key)
        .key_type(KeyType::Hash)
        .build()
        .expect("creating KeySchemaElement");

    match client
        .create_table()
        .table_name(table)
        .key_schema(ks)
        .attribute_definitions(ad)
        .billing_mode(BillingMode::PayPerRequest)
        .send()
        .await
    {
        Ok(_) => Ok(()),
        Err(e) => Err(e),
    }
}
```

```
async fn add_item(client: &Client, item: Item) -> Result<(),
 SdkError<ExecuteStatementError>> {
    match client
        .execute_statement()
        .statement(format!(
            r#"INSERT INTO "{}" VALUE {{
                "{}": ?,
                "acount_type": ?,
                "age": ?,
                "first_name": ?,
                "last_name": ?
        }} "#,
            item.table, item.key
        ))
        .set_parameters(Some(vec![
            AttributeValue::S(item.utype),
            AttributeValue::S(item.age),
            AttributeValue::S(item.first_name),
            AttributeValue::S(item.last_name),
        ]))
        .send()
        .await
    {
        Ok(_) => Ok(()),
        Err(e) => Err(e),
    }
}

async fn query_item(client: &Client, item: Item) -> bool {
    match client
        .execute_statement()
        .statement(format!(
            r#"SELECT * FROM "{}" WHERE "{}" = ?"#,
            item.table, item.key
        ))
        .set_parameters(Some(vec![AttributeValue::S(item.value)]))
        .send()
        .await
    {
        Ok(resp) => {
            if !resp.items().is_empty() {
                println!("Found a matching entry in the table:");
                println!("{:?}", resp.items.unwrap_or_default().pop());
                true
```

```
            } else {
                println!("Did not find a match.");
                false
            }
        }
        Err(e) => {
            println!("Got an error querying table:");
            println!("{}", e);
            process::exit(1);
        }
    }
}

async fn remove_item(client: &Client, table: &str, key: &str, value: String) ->
 Result<(), Error> {
    client
        .execute_statement()
        .statement(format!(r#"DELETE FROM "{table}" WHERE "{key}" = ?"#))
        .set_parameters(Some(vec![AttributeValue::S(value)]))
        .send()
        .await?;

    println!("Deleted item.");

    Ok(())
}

async fn remove_table(client: &Client, table: &str) -> Result<(), Error> {
    client.delete_table().table_name(table).send().await?;

    Ok(())
}
```

- For API details, see ExecuteStatement in *AWS SDK for Rust API reference*.

For a complete list of AWS SDK developer guides and code examples, see Using DynamoDB with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Query a DynamoDB table using a Global Secondary Index with an AWS SDK

The following code examples show how to query a table using a Global Secondary Index.

- Query a DynamoDB table using its primary key.
- Query a Global Secondary Index (GSI) for alternate access patterns.
- Compare table queries and GSI queries.

Java

SDK for Java 2.x

Query a DynamoDB table using its primary key and a Global Secondary Index (GSI) with AWS SDK for Java 2.x.

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.QueryRequest;
import software.amazon.awssdk.services.dynamodb.model.QueryResponse;
import software.amazon.awssdk.services.dynamodb.model.ResourceNotFoundException;

import java.util.HashMap;
import java.util.Map;

    public QueryResponse queryTable(
        final String tableName, final String partitionKeyName, final String
 partitionKeyValue) {

        CodeSampleUtils.validateTableParameters(tableName, partitionKeyName,
 partitionKeyValue);

        // Create expression attribute names for the column names
        final Map<String, String> expressionAttributeNames = new HashMap<>();
        expressionAttributeNames.put(EXPRESSION_ATTRIBUTE_NAME_PK,
 partitionKeyName);

        // Create expression attribute values for the column values
```

```
        final Map<String, AttributeValue> expressionAttributeValues = new
HashMap<>();
        expressionAttributeValues.put(
            EXPRESSION_ATTRIBUTE_VALUE_PK,
            AttributeValue.builder().s(partitionKeyValue).build());

        // Create the query request
        final QueryRequest queryRequest = QueryRequest.builder()
            .tableName(tableName)
            .keyConditionExpression(KEY_CONDITION_EXPRESSION)
            .expressionAttributeNames(expressionAttributeNames)
            .expressionAttributeValues(expressionAttributeValues)
            .build();

        try {
            final QueryResponse response = dynamoDbClient.query(queryRequest);
            System.out.println("Query on base table successful. Found " +
response.count() + " items");
            return response;
        } catch (ResourceNotFoundException e) {
            System.err.format("Error: The Amazon DynamoDB table \"%s\" can't be
found.\n", tableName);
            throw new DynamoDbQueryException("Table not found: " + tableName, e);
        } catch (DynamoDbException e) {
            System.err.println("Error querying base table: " + e.getMessage());
            throw new DynamoDbQueryException("Failed to execute query on base
table", e);
        }
    }

    /**
     * Queries a DynamoDB Global Secondary Index (GSI) by partition key.
     *
     * @param tableName         The name of the DynamoDB table
     * @param indexName         The name of the GSI
     * @param partitionKeyName  The name of the GSI partition key attribute
     * @param partitionKeyValue The value of the GSI partition key to query
     * @return The query response from DynamoDB
     * @throws ResourceNotFoundException if the table or index doesn't exist
     * @throws DynamoDbException if the query fails
     */
    public QueryResponse queryGlobalSecondaryIndex(
        final String tableName, final String indexName, final String
partitionKeyName, final String partitionKeyValue) {
```

```
       CodeSampleUtils.validateTableParameters(tableName, partitionKeyName,
partitionKeyValue);
       CodeSampleUtils.validateStringParameter("Index name", indexName);

       // Create expression attribute names for the column names
       final Map<String, String> expressionAttributeNames = new HashMap<>();
       expressionAttributeNames.put(EXPRESSION_ATTRIBUTE_NAME_IK,
partitionKeyName);

       // Create expression attribute values for the column values
       final Map<String, AttributeValue> expressionAttributeValues = new
HashMap<>();
       expressionAttributeValues.put(
           EXPRESSION_ATTRIBUTE_VALUE_IK,
           AttributeValue.builder().s(partitionKeyValue).build());

       // Create the query request
       final QueryRequest queryRequest = QueryRequest.builder()
           .tableName(tableName)
           .indexName(indexName)
           .keyConditionExpression(GSI_KEY_CONDITION_EXPRESSION)
           .expressionAttributeNames(expressionAttributeNames)
           .expressionAttributeValues(expressionAttributeValues)
           .build();

       try {
           final QueryResponse response = dynamoDbClient.query(queryRequest);
           System.out.println("Query on GSI successful. Found " +
response.count() + " items");
           return response;
       } catch (ResourceNotFoundException e) {
           System.err.format(
               "Error: The Amazon DynamoDB table \"%s\" or index \"%s\" can't be
found.\n", tableName, indexName);
           throw new DynamoDbQueryException("Table or index not found: " +
tableName + "/" + indexName, e);
       } catch (DynamoDbException e) {
           System.err.println("Error querying GSI: " + e.getMessage());
           throw new DynamoDbQueryException("Failed to execute query on GSI",
e);
       }
   }
```

Compare querying a table directly versus querying a GSI with AWS SDK for Java 2.x.

```
   public static void main(String[] args) {
       final String usage =
           """
               Usage:
                   <tableName> <basePartitionKeyName> <basePartitionKeyValue>
<gsiName> <gsiPartitionKeyName> <gsiPartitionKeyValue> [region]
               Where:
                   tableName - The Amazon DynamoDB table to query.
                   basePartitionKeyName - The name of the base table partition
key attribute.
                   basePartitionKeyValue - The value of the base table partition
key to query.
                   gsiName - The name of the Global Secondary Index.
                   gsiPartitionKeyName - The name of the GSI partition key
attribute.
                   gsiPartitionKeyValue - The value of the GSI partition key to
query.
                   region (optional) - The AWS region where the table exists.
(Default: us-east-1)
               """;

       if (args.length < 6) {
           System.out.println(usage);
           System.exit(1);
       }

       final String tableName = args[0];
       final String basePartitionKeyName = args[1];
       final String basePartitionKeyValue = args[2];
       final String gsiName = args[3];
       final String gsiPartitionKeyName = args[4];
       final String gsiPartitionKeyValue = args[5];
       final Region region = args.length > 6 ? Region.of(args[6]) :
Region.US_EAST_1;

       try (DynamoDbClient ddb =
DynamoDbClient.builder().region(region).build()) {
           final QueryTableAndGSI queryHelper = new QueryTableAndGSI(ddb);
```

```
            // Query the base table
            System.out.println("Querying base table where " +
basePartitionKeyName + " = " + basePartitionKeyValue);
            final QueryResponse tableResponse =
                queryHelper.queryTable(tableName, basePartitionKeyName,
basePartitionKeyValue);

            System.out.println("Found " + tableResponse.count() + " items in base
table:");
            tableResponse.items().forEach(item -> System.out.println(item));

            // Query the GSI
            System.out.println(
                "\nQuerying GSI '" + gsiName + "' where " + gsiPartitionKeyName +
" = " + gsiPartitionKeyValue);
            final QueryResponse gsiResponse =
                queryHelper.queryGlobalSecondaryIndex(tableName, gsiName,
gsiPartitionKeyName, gsiPartitionKeyValue);

            System.out.println("Found " + gsiResponse.count() + " items in
GSI:");
            gsiResponse.items().forEach(item -> System.out.println(item));

            // Explain the differences between querying a table and a GSI
            System.out.println("\nKey differences between querying a table and a
GSI:");
            System.out.println("1. When querying a GSI, you must specify the
indexName parameter");
            System.out.println("2. GSIs may not contain all attributes from the
base table (projection)");
            System.out.println("3. GSIs consume read capacity units from the
GSI's capacity, not the base table's");
            System.out.println("4. GSIs may have eventually consistent data
(cannot use ConsistentRead=true)");

        } catch (IllegalArgumentException e) {
            System.err.println("Invalid input: " + e.getMessage());
            System.exit(1);
        } catch (ResourceNotFoundException e) {
            System.err.println("Table or index not found: " + e.getMessage());
            System.exit(1);
        } catch (DynamoDbException e) {
            System.err.println("DynamoDB error: " + e.getMessage());
            System.exit(1);
```

```
        } catch (Exception e) {
            System.err.println("Unexpected error: " + e.getMessage());
            System.exit(1);
        }
    }
```

- For API details, see Query in *AWS SDK for Java 2.x API Reference*.

JavaScript

SDK for JavaScript (v3)

Query a DynamoDB table using the primary key with AWS SDK for JavaScript.

```
const { DynamoDBClient, QueryCommand } = require("@aws-sdk/client-dynamodb");

/**
 * Queries a DynamoDB table using the primary key
 *
 * @param {Object} config - AWS SDK configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {string} userId - The user ID to query by (partition key)
 * @returns {Promise<Object>} - The query response
 */
async function queryTable(
  config,
  tableName,
  userId
) {
  try {
    // Create DynamoDB client
    const client = new DynamoDBClient(config);

    // Construct the query input for the base table
    const input = {
      TableName: tableName,
      KeyConditionExpression: "user_id = :userId",
      ExpressionAttributeValues: {
        ":userId": { S: userId }
      }
    };
```

```
    // Execute the query
    const command = new QueryCommand(input);
    return await client.send(command);
  } catch (error) {
    console.error(`Error querying table: ${error}`);
    throw error;
  }
}
```

Query a DynamoDB Global Secondary Index (GSI) with AWS SDK for JavaScript.

```
/**
 * Queries a DynamoDB Global Secondary Index (GSI)
 *
 * @param {Object} config - AWS SDK configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {string} indexName - The name of the GSI to query
 * @param {string} gameId - The game ID to query by (GSI partition key)
 * @returns {Promise<Object>} - The query response
 */
async function queryGSI(
  config,
  tableName,
  indexName,
  gameId
) {
  try {
    // Create DynamoDB client
    const client = new DynamoDBClient(config);

    // Construct the query input for the GSI
    const input = {
      TableName: tableName,
      IndexName: indexName,
      KeyConditionExpression: "game_id = :gameId",
      ExpressionAttributeValues: {
        ":gameId": { S: gameId }
      }
    };

    // Execute the query
    const command = new QueryCommand(input);
```

```
    return await client.send(command);
  } catch (error) {
    console.error(`Error querying GSI: ${error}`);
    throw error;
  }
}
```

- For API details, see Query in *AWS SDK for JavaScript API Reference*.

Python

SDK for Python (Boto3)

Query a DynamoDB table using its primary key and a Global Secondary Index (GSI) with AWS SDK for Python (Boto3).

```
import boto3
from boto3.dynamodb.conditions import Key


def query_table(table_name, partition_key_name, partition_key_value):
    """
    Query a DynamoDB table using its primary key.

    Args:
        table_name (str): The name of the DynamoDB table.
        partition_key_name (str): The name of the partition key attribute.
        partition_key_value (str): The value of the partition key to query.

    Returns:
        dict: The response from DynamoDB containing the query results.
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # Perform the query on the table's primary key
    response =
 table.query(KeyConditionExpression=Key(partition_key_name).eq(partition_key_value))

    return response
```

```
def query_gsi(table_name, index_name, partition_key_name, partition_key_value):
    """
    Query a Global Secondary Index (GSI) on a DynamoDB table.

    Args:
        table_name (str): The name of the DynamoDB table.
        index_name (str): The name of the Global Secondary Index.
        partition_key_name (str): The name of the GSI's partition key attribute.
        partition_key_value (str): The value of the GSI's partition key to query.

    Returns:
        dict: The response from DynamoDB containing the query results.
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # Perform the query on the GSI
    response = table.query(
        IndexName=index_name,
 KeyConditionExpression=Key(partition_key_name).eq(partition_key_value)
    )

    return response
```

- For API details, see Query in *AWS SDK for Python (Boto3) API Reference*.

For a complete list of AWS SDK developer guides and code examples, see Using DynamoDB with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Query a DynamoDB table using a begins_with condition with an AWS SDK

The following code examples show how to query a table using a begins_with condition.

- Use the begins_with function in a key condition expression.

- Filter items based on a prefix pattern in the sort key.

Java

SDK for Java 2.x

Query a DynamoDB table using a begins_with condition on the sort key with AWS SDK for Java 2.x.

```
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.QueryRequest;
import software.amazon.awssdk.services.dynamodb.model.QueryResponse;
import software.amazon.awssdk.services.dynamodb.model.ResourceNotFoundException;

import java.util.HashMap;
import java.util.Map;
import java.util.logging.Level;
import java.util.logging.Logger;

    public QueryResponse queryWithBeginsWithCondition(
        final String tableName,
        final String partitionKeyName,
        final String partitionKeyValue,
        final String sortKeyName,
        final String sortKeyPrefix) {

        CodeSampleUtils.validateTableParameters(tableName, partitionKeyName,
 partitionKeyValue);
        CodeSampleUtils.validateStringParameter("Sort key name", sortKeyName);
        CodeSampleUtils.validateStringParameter("Sort key prefix",
 sortKeyPrefix);

        // Create expression attribute names for the column names
        final Map<String, String> expressionAttributeNames = new HashMap<>();
        expressionAttributeNames.put(EXPRESSION_ATTRIBUTE_NAME_PK,
 partitionKeyName);
        expressionAttributeNames.put(EXPRESSION_ATTRIBUTE_NAME_SK, sortKeyName);

        // Create expression attribute values for the column values
        final Map<String, AttributeValue> expressionAttributeValues = new
 HashMap<>();
```

```
        expressionAttributeValues.put(
            EXPRESSION_ATTRIBUTE_VALUE_PK,
            AttributeValue.builder().s(partitionKeyValue).build());
        expressionAttributeValues.put(
            EXPRESSION_ATTRIBUTE_VALUE_SK_PREFIX,
            AttributeValue.builder().s(sortKeyPrefix).build());

        // Create the query request
        final QueryRequest queryRequest = QueryRequest.builder()
            .tableName(tableName)
            .keyConditionExpression(KEY_CONDITION_EXPRESSION)
            .expressionAttributeNames(expressionAttributeNames)
            .expressionAttributeValues(expressionAttributeValues)
            .build();

        try {
            final QueryResponse response = dynamoDbClient.query(queryRequest);
            LOGGER.log(Level.INFO, "Query with begins_with condition successful.
Found {0} items", response.count());
            return response;
        } catch (ResourceNotFoundException e) {
            LOGGER.log(Level.SEVERE, "Table not found: {0}", tableName);
            throw e;
        } catch (DynamoDbException e) {
            LOGGER.log(Level.SEVERE, "Error querying with begins_with condition",
e);
            throw e;
        }
    }
```

Demonstrate using begins_with with different prefix lengths with AWS SDK for Java 2.x.

```
    public static void main(String[] args) {
        try {
            CodeSampleUtils.BeginsWithQueryConfig config =
CodeSampleUtils.BeginsWithQueryConfig.fromArgs(args);
            LOGGER.log(Level.INFO, "Querying items where {0} = {1} and {2} begins
with ''{3}''", new Object[] {
                config.getPartitionKeyName(),
                config.getPartitionKeyValue(),
                config.getSortKeyName(),
                config.getSortKeyPrefix()
```

```
            });

            // Using the builder pattern to create and execute the query
            final QueryResponse response = new BeginsWithQueryBuilder()
                .withTableName(config.getTableName())
                .withPartitionKeyName(config.getPartitionKeyName())
                .withPartitionKeyValue(config.getPartitionKeyValue())
                .withSortKeyName(config.getSortKeyName())
                .withSortKeyPrefix(config.getSortKeyPrefix())
                .withRegion(config.getRegion())
                .execute();

            // Process the results
            LOGGER.log(Level.INFO, "Found {0} items:", response.count());
            response.items().forEach(item -> LOGGER.info(item.toString()));

            // Demonstrate with a different prefix
            if (!config.getSortKeyPrefix().isEmpty()) {
                String shorterPrefix = config.getSortKeyPrefix()
                    .substring(0, Math.max(1,
config.getSortKeyPrefix().length() / 2));
                LOGGER.log(Level.INFO, "\nNow querying with a shorter prefix:
''{0}''", shorterPrefix);

                final QueryResponse response2 = new BeginsWithQueryBuilder()
                    .withTableName(config.getTableName())
                    .withPartitionKeyName(config.getPartitionKeyName())
                    .withPartitionKeyValue(config.getPartitionKeyValue())
                    .withSortKeyName(config.getSortKeyName())
                    .withSortKeyPrefix(shorterPrefix)
                    .withRegion(config.getRegion())
                    .execute();

                LOGGER.log(Level.INFO, "Found {0} items with shorter prefix:",
response2.count());
                response2.items().forEach(item -> LOGGER.info(item.toString()));
            }
        } catch (IllegalArgumentException e) {
            LOGGER.log(Level.SEVERE, "Invalid input: {0}", e.getMessage());
            printUsage();
        } catch (ResourceNotFoundException e) {
            LOGGER.log(Level.SEVERE, "Table not found", e);
        } catch (DynamoDbException e) {
            LOGGER.log(Level.SEVERE, "DynamoDB error", e);
```

```
        } catch (Exception e) {
            LOGGER.log(Level.SEVERE, "Unexpected error", e);
        }
    }
```

- For API details, see Query in *AWS SDK for Java 2.x API Reference*.

JavaScript

SDK for JavaScript (v3)

Query a DynamoDB table using a begins_with condition on the sort key with AWS SDK for JavaScript.

```
const { DynamoDBClient, QueryCommand } = require("@aws-sdk/client-dynamodb");

/**
 * Queries a DynamoDB table for items where the sort key begins with a specific
 prefix
 *
 * @param {Object} config - AWS SDK configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {string} partitionKeyName - The name of the partition key
 * @param {string} partitionKeyValue - The value of the partition key
 * @param {string} sortKeyName - The name of the sort key
 * @param {string} prefix - The prefix to match at the beginning of the sort key
 * @returns {Promise<Object>} - The query response
 */
async function queryWithBeginsWith(
  config,
  tableName,
  partitionKeyName,
  partitionKeyValue,
  sortKeyName,
  prefix
) {
  try {
    // Create DynamoDB client
    const client = new DynamoDBClient(config);

    // Construct the query input
    const input = {
```

```
      TableName: tableName,
      KeyConditionExpression: "#pk = :pkValue AND begins_with(#sk, :prefix)",
      ExpressionAttributeNames: {
        "#pk": partitionKeyName,
        "#sk": sortKeyName
      },
      ExpressionAttributeValues: {
        ":pkValue": { S: partitionKeyValue },
        ":prefix": { S: prefix }
      }
    };

    // Execute the query
    const command = new QueryCommand(input);
    return await client.send(command);
  } catch (error) {
    console.error(`Error querying with begins_with: ${error}`);
    throw error;
  }
}
```

- For API details, see Query in *AWS SDK for JavaScript API Reference*.

Python

SDK for Python (Boto3)

Query a DynamoDB table using a begins_with condition on the sort key with AWS SDK for Python (Boto3).

```
import boto3
from boto3.dynamodb.conditions import Key


def query_with_begins_with(
    table_name, partition_key_name, partition_key_value, sort_key_name, prefix
):
    """
    Query a DynamoDB table with a begins_with condition on the sort key.

    Args:
        table_name (str): The name of the DynamoDB table.
```

```
        partition_key_name (str): The name of the partition key attribute.
        partition_key_value (str): The value of the partition key to query.
        sort_key_name (str): The name of the sort key attribute.
        prefix (str): The prefix to match at the beginning of the sort key.

    Returns:
        dict: The response from DynamoDB containing the query results.
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # Perform the query with a begins_with condition on the sort key
    key_condition = Key(partition_key_name).eq(partition_key_value) & Key(
        sort_key_name
    ).begins_with(prefix)
    response = table.query(KeyConditionExpression=key_condition)

    return response
```

- For API details, see Query in *AWS SDK for Python (Boto3) API Reference*.

For a complete list of AWS SDK developer guides and code examples, see Using DynamoDB with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Query a DynamoDB table using a date range in the sort key with an AWS SDK

The following code examples show how to query a table using a date range in the sort key.

- Query items within a specific date range.
- Use comparison operators on date-formatted sort keys.

Java

SDK for Java 2.x

Query a DynamoDB table for items within a date range with AWS SDK for Java 2.x.

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.QueryRequest;
import software.amazon.awssdk.services.dynamodb.model.QueryResponse;
import software.amazon.awssdk.services.dynamodb.model.ResourceNotFoundException;

import java.time.LocalDate;
import java.util.HashMap;
import java.util.Map;
import java.util.logging.Level;
import java.util.logging.Logger;

    public QueryResponse queryWithDateRange(
        final String tableName,
        final String partitionKeyName,
        final String partitionKeyValue,
        final String dateKeyName,
        final LocalDate startDate,
        final LocalDate endDate) {

        // Focus on query logic, assuming parameters are valid
        if (startDate == null || endDate == null) {
            throw new IllegalArgumentException("Start date and end date cannot be
 null");
        }

        if (endDate.isBefore(startDate)) {
            throw new IllegalArgumentException("End date must be after start
 date");
        }

        // Format dates as ISO strings for DynamoDB (using just the date part)
        final String formattedStartDate = startDate.toString();
        final String formattedEndDate = endDate.toString();

        // Create expression attribute names for the column names
```

```
        final Map<String, String> expressionAttributeNames = new HashMap<>();
        expressionAttributeNames.put(EXPRESSION_ATTRIBUTE_NAME_PK,
partitionKeyName);
        expressionAttributeNames.put(EXPRESSION_ATTRIBUTE_NAME_SK, dateKeyName);

        // Create expression attribute values for the column values
        final Map<String, AttributeValue> expressionAttributeValues = new
HashMap<>();
        expressionAttributeValues.put(
            EXPRESSION_ATTRIBUTE_VALUE_PK,
            AttributeValue.builder().s(partitionKeyValue).build());
        expressionAttributeValues.put(
            EXPRESSION_ATTRIBUTE_VALUE_START_DATE,
            AttributeValue.builder().s(formattedStartDate).build());
        expressionAttributeValues.put(
            EXPRESSION_ATTRIBUTE_VALUE_END_DATE,
            AttributeValue.builder().s(formattedEndDate).build());

        // Create the query request
        final QueryRequest queryRequest = QueryRequest.builder()
            .tableName(tableName)
            .keyConditionExpression(KEY_CONDITION_EXPRESSION)
            .expressionAttributeNames(expressionAttributeNames)
            .expressionAttributeValues(expressionAttributeValues)
            .build();

        try {
            final QueryResponse response = dynamoDbClient.query(queryRequest);
            LOGGER.log(Level.INFO, "Query by date range successful. Found {0}
items", response.count());
            return response;
        } catch (ResourceNotFoundException e) {
            LOGGER.log(Level.SEVERE, "Table not found: {0}", tableName);
            throw e;
        } catch (DynamoDbException e) {
            LOGGER.log(Level.SEVERE, "Error querying by date range: {0}",
e.getMessage());
            throw e;
        }
    }
```

Demonstrates how to query a DynamoDB table with date range filtering.

```
   public static void main(String[] args) {
       final String usage =
           """
               Usage:
                   <tableName> <partitionKeyName> <partitionKeyValue>
<dateKeyName> <startDate> <endDate> [region]
               Where:
                   tableName - The Amazon DynamoDB table to query.
                   partitionKeyName - The name of the partition key attribute.
                   partitionKeyValue - The value of the partition key to query.
                   dateKeyName - The name of the date attribute to filter on.
                   startDate - The start date for the range query (YYYY-MM-DD).
                   endDate - The end date for the range query (YYYY-MM-DD).
                   region (optional) - The AWS region where the table exists.
(Default: us-east-1)
               """;

       if (args.length < 6) {
           System.out.println(usage);
           System.exit(1);
       }

       try {
           // Parse command line arguments into a config object
           CodeSampleUtils.DateRangeQueryConfig config =
CodeSampleUtils.DateRangeQueryConfig.fromArgs(args);

           LOGGER.log(
               Level.INFO, "Querying items from {0} to {1}", new Object[]
{config.getStartDate(), config.getEndDate()
               });

           // Using the builder pattern to create and execute the query
           final QueryResponse response = new DateRangeQueryBuilder()
               .withTableName(config.getTableName())
               .withPartitionKeyName(config.getPartitionKeyName())
               .withPartitionKeyValue(config.getPartitionKeyValue())
               .withDateKeyName(config.getDateKeyName())
               .withStartDate(config.getStartDate())
               .withEndDate(config.getEndDate())
               .withRegion(config.getRegion())
               .execute();
```

```
            // Process the results
            LOGGER.log(Level.INFO, "Found {0} items:", response.count());
            response.items().forEach(item -> {
                LOGGER.info(item.toString());

                // Extract and display the date attribute for clarity
                if (item.containsKey(config.getDateKeyName())) {
                    LOGGER.log(
                        Level.INFO,
                        "  Date attribute: {0}",
                        item.get(config.getDateKeyName()).s());
                }
            });

            // Demonstrate with a different date range
            LocalDate narrowerStartDate = config.getStartDate().plusDays(1);
            LocalDate narrowerEndDate = config.getEndDate().minusDays(1);

            if (!narrowerStartDate.isAfter(narrowerEndDate)) {
                LOGGER.log(Level.INFO, "\nNow querying with a narrower date
range: {0} to {1}", new Object[] {
                    narrowerStartDate, narrowerEndDate
                });

                final QueryResponse response2 = new DateRangeQueryBuilder()
                    .withTableName(config.getTableName())
                    .withPartitionKeyName(config.getPartitionKeyName())
                    .withPartitionKeyValue(config.getPartitionKeyValue())
                    .withDateKeyName(config.getDateKeyName())
                    .withStartDate(narrowerStartDate)
                    .withEndDate(narrowerEndDate)
                    .withRegion(config.getRegion())
                    .execute();

                LOGGER.log(Level.INFO, "Found {0} items with narrower date
range:", response2.count());
                response2.items().forEach(item -> LOGGER.info(item.toString()));
            }

            LOGGER.info("\nNote: When storing dates in DynamoDB:");
            LOGGER.info("1. Use ISO format (YYYY-MM-DD) for lexicographical
ordering");
            LOGGER.info("2. Use the BETWEEN operator for inclusive date range
queries");
```

```
            LOGGER.info("3. Consider using ISO-8601 format for timestamps with
 time components");

        } catch (IllegalArgumentException e) {
            LOGGER.log(Level.SEVERE, "Invalid input: {0}", e.getMessage());
            System.exit(1);
        } catch (ResourceNotFoundException e) {
            LOGGER.log(Level.SEVERE, "Table not found: {0}", e.getMessage());
            System.exit(1);
        } catch (DynamoDbException e) {
            LOGGER.log(Level.SEVERE, "DynamoDB error: {0}", e.getMessage());
            System.exit(1);
        } catch (Exception e) {
            LOGGER.log(Level.SEVERE, "Unexpected error: {0}", e.getMessage());
            System.exit(1);
        }
    }
```

- For API details, see Query in *AWS SDK for Java 2.x API Reference*.

JavaScript

SDK for JavaScript (v3)

Query a DynamoDB table for items within a date range with AWS SDK for JavaScript.

```
const { DynamoDBClient, QueryCommand } = require("@aws-sdk/client-dynamodb");

/**
 * Queries a DynamoDB table for items within a specific date range on the sort
 key
 *
 * @param {Object} config - AWS SDK configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {string} partitionKeyName - The name of the partition key
 * @param {string} partitionKeyValue - The value of the partition key
 * @param {string} sortKeyName - The name of the sort key (must be a date/time
 attribute)
 * @param {Date} startDate - The start date for the range query
 * @param {Date} endDate - The end date for the range query
 * @returns {Promise<Object>} - The query response
 */
```

```
async function queryByDateRangeOnSortKey(
  config,
  tableName,
  partitionKeyName,
  partitionKeyValue,
  sortKeyName,
  startDate,
  endDate
) {
  try {
    // Create DynamoDB client
    const client = new DynamoDBClient(config);

    // Format dates as ISO strings for DynamoDB
    const formattedStartDate = startDate.toISOString();
    const formattedEndDate = endDate.toISOString();

    // Construct the query input
    const input = {
      TableName: tableName,
      KeyConditionExpression: '#pk = :pkValue AND #sk BETWEEN :startDate
 AND :endDate',
      ExpressionAttributeNames: {
        "#pk": partitionKeyName,
        "#sk": sortKeyName
      },
      ExpressionAttributeValues: {
        ":pkValue": { S: partitionKeyValue },
        ":startDate": { S: formattedStartDate },
        ":endDate": { S: formattedEndDate }
      }
    };

    // Execute the query
    const command = new QueryCommand(input);
    return await client.send(command);
  } catch (error) {
    console.error(`Error querying by date range on sort key: ${error}`);
    throw error;
  }
}
```

- For API details, see Query in *AWS SDK for JavaScript API Reference*.

Python

SDK for Python (Boto3)

Query a DynamoDB table for items within a date range with AWS SDK for Python (Boto3).

```
from datetime import datetime, timedelta

import boto3
from boto3.dynamodb.conditions import Key


def query_with_date_range(
    table_name, partition_key_name, partition_key_value, sort_key_name,
 start_date, end_date
):
    """
    Query a DynamoDB table with a date range on the sort key.

    Args:
        table_name (str): The name of the DynamoDB table.
        partition_key_name (str): The name of the partition key attribute.
        partition_key_value (str): The value of the partition key to query.
        sort_key_name (str): The name of the sort key attribute (containing date
 values).
        start_date (datetime): The start date for the query range.
        end_date (datetime): The end date for the query range.

    Returns:
        dict: The response from DynamoDB containing the query results.
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # Format the date values as ISO 8601 strings
    # DynamoDB works well with ISO format for date values
    start_date_str = start_date.isoformat()
    end_date_str = end_date.isoformat()

    # Perform the query with a date range on the sort key using BETWEEN operator
    key_condition = Key(partition_key_name).eq(partition_key_value) &
 Key(sort_key_name).between(
        start_date_str, end_date_str
```

```
    )

    response = table.query(
        KeyConditionExpression=key_condition,
        ExpressionAttributeValues={
            ":pk_val": partition_key_value,
            ":start_date": start_date_str,
            ":end_date": end_date_str,
        },
    )

    return response


def query_with_date_range_by_month(
    table_name, partition_key_name, partition_key_value, sort_key_name, year,
 month
):
    """
    Query a DynamoDB table for a specific month's data.

    Args:
        table_name (str): The name of the DynamoDB table.
        partition_key_name (str): The name of the partition key attribute.
        partition_key_value (str): The value of the partition key to query.
        sort_key_name (str): The name of the sort key attribute (containing date
 values).
        year (int): The year to query.
        month (int): The month to query (1-12).

    Returns:
        dict: The response from DynamoDB containing the query results.
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # Calculate the start and end dates for the specified month
    if month == 12:
        next_year = year + 1
        next_month = 1
    else:
        next_year = year
        next_month = month + 1
```

```
    start_date = datetime(year, month, 1)
    end_date = datetime(next_year, next_month, 1) - timedelta(microseconds=1)

    # Format the date values as ISO 8601 strings
    start_date_str = start_date.isoformat()
    end_date_str = end_date.isoformat()

    # Perform the query with a date range on the sort key
    key_condition = Key(partition_key_name).eq(partition_key_value) &
Key(sort_key_name).between(
        start_date_str, end_date_str
    )

    response = table.query(KeyConditionExpression=key_condition)

    return response
```

- For API details, see Query in *AWS SDK for Python (Boto3) API Reference*.

For a complete list of AWS SDK developer guides and code examples, see Using DynamoDB with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Query a DynamoDB table with a complex filter expression with an AWS SDK

The following code examples show how to query a table with a complex filter expression.

- Apply complex filter expressions to query results.
- Combine multiple conditions using logical operators.
- Filter items based on non-key attributes.

Java

SDK for Java 2.x

Query a DynamoDB table with a complex filter expression using AWS SDK for Java 2.x.

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.QueryRequest;
import software.amazon.awssdk.services.dynamodb.model.QueryResponse;
import software.amazon.awssdk.services.dynamodb.model.ResourceNotFoundException;

import java.util.HashMap;
import java.util.Map;
import java.util.logging.Level;
import java.util.logging.Logger;

    public QueryResponse queryWithComplexFilter(
        final String tableName,
        final String partitionKeyName,
        final String partitionKeyValue,
        final String statusAttrName,
        final String activeStatus,
        final String pendingStatus,
        final String priceAttrName,
        final double minPrice,
        final double maxPrice,
        final String categoryAttrName) {

        // Validate parameters
        CodeSampleUtils.validateTableParameters(tableName, partitionKeyName,
 partitionKeyValue);
       CodeSampleUtils.validateStringParameter("Status attribute name",
 statusAttrName);
       CodeSampleUtils.validateStringParameter("Active status", activeStatus);
       CodeSampleUtils.validateStringParameter("Pending status", pendingStatus);
       CodeSampleUtils.validateStringParameter("Price attribute name",
 priceAttrName);
       CodeSampleUtils.validateStringParameter("Category attribute name",
 categoryAttrName);
       CodeSampleUtils.validateNumericRange("Minimum price", minPrice, 0.0,
 Double.MAX_VALUE);
```

```
        CodeSampleUtils.validateNumericRange("Maximum price", maxPrice, minPrice,
Double.MAX_VALUE);

        // Create expression attribute names for the column names
        final Map<String, String> expressionAttributeNames = new HashMap<>();
        expressionAttributeNames.put("#pk", partitionKeyName);
        expressionAttributeNames.put(EXPRESSION_ATTRIBUTE_NAME_STATUS,
statusAttrName);
        expressionAttributeNames.put(EXPRESSION_ATTRIBUTE_NAME_PRICE,
priceAttrName);
        expressionAttributeNames.put(EXPRESSION_ATTRIBUTE_NAME_CATEGORY,
categoryAttrName);

        // Create expression attribute values for the column values
        final Map<String, AttributeValue> expressionAttributeValues = new
HashMap<>();
        expressionAttributeValues.put(
            ":pkValue", AttributeValue.builder().s(partitionKeyValue).build());
        expressionAttributeValues.put(
            EXPRESSION_ATTRIBUTE_VALUE_ACTIVE,
            AttributeValue.builder().s(activeStatus).build());
        expressionAttributeValues.put(
            EXPRESSION_ATTRIBUTE_VALUE_PENDING,
            AttributeValue.builder().s(pendingStatus).build());
        expressionAttributeValues.put(
            EXPRESSION_ATTRIBUTE_VALUE_MIN_PRICE,
            AttributeValue.builder().n(String.valueOf(minPrice)).build());
        expressionAttributeValues.put(
            EXPRESSION_ATTRIBUTE_VALUE_MAX_PRICE,
            AttributeValue.builder().n(String.valueOf(maxPrice)).build());

        // Create the query request
        final QueryRequest queryRequest = QueryRequest.builder()
            .tableName(tableName)
            .keyConditionExpression(KEY_CONDITION_EXPRESSION)
            .filterExpression(FILTER_EXPRESSION)
            .expressionAttributeNames(expressionAttributeNames)
            .expressionAttributeValues(expressionAttributeValues)
            .build();

        return dynamoDbClient.query(queryRequest);
    }
```

- For API details, see Query in *AWS SDK for Java 2.x API Reference*.

JavaScript

SDK for JavaScript (v3)

Query a DynamoDB table with a complex filter expression using AWS SDK for JavaScript.

```
const { DynamoDBClient, QueryCommand } = require("@aws-sdk/client-dynamodb");

/**
 * Queries a DynamoDB table with a complex filter expression
 *
 * @param {Object} config - AWS SDK configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {string} partitionKeyName - The name of the partition key
 * @param {string} partitionKeyValue - The value of the partition key
 * @param {number|string} minViews - Minimum number of views for filtering
 * @param {number|string} minReplies - Minimum number of replies for filtering
 * @param {string} requiredTag - Tag that must be present in the item's tags set
 * @returns {Promise<Object>} - The query response
 */
async function queryWithComplexFilter(
  config,
  tableName,
  partitionKeyName,
  partitionKeyValue,
  minViews,
  minReplies,
  requiredTag
) {
  try {
    // Create DynamoDB client
    const client = new DynamoDBClient(config);

    // Construct the query input
    const input = {
      TableName: tableName,
      KeyConditionExpression: "#pk = :pkValue",
      FilterExpression: "views >= :minViews AND replies >= :minReplies AND
 contains(tags, :tag)",
      ExpressionAttributeNames: {
        "#pk": partitionKeyName
```

```
      },
      ExpressionAttributeValues: {
        ":pkValue": { S: partitionKeyValue },
        ":minViews": { N: minViews.toString() },
        ":minReplies": { N: minReplies.toString() },
        ":tag": { S: requiredTag }
      }
    };

    // Execute the query
    const command = new QueryCommand(input);
    return await client.send(command);
  } catch (error) {
    console.error(`Error querying with complex filter: ${error}`);
    throw error;
  }
}
```

- For API details, see Query in *AWS SDK for JavaScript API Reference*.

Python

SDK for Python (Boto3)

Query a DynamoDB table with a complex filter expression using AWS SDK for Python (Boto3).

```
import boto3
from boto3.dynamodb.conditions import Attr, Key


def query_with_complex_filter(
    table_name,
    partition_key_name,
    partition_key_value,
    min_rating=None,
    status_list=None,
    max_price=None,
):
    """
    Query a DynamoDB table with a complex filter expression.
```

```
    Args:
        table_name (str): The name of the DynamoDB table.
        partition_key_name (str): The name of the partition key attribute.
        partition_key_value (str): The value of the partition key to query.
        min_rating (float, optional): Minimum rating value for filtering.
        status_list (list, optional): List of status values to include.
        max_price (float, optional): Maximum price value for filtering.

    Returns:
        dict: The response from DynamoDB containing the query results.
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # Start with the key condition expression
    key_condition = Key(partition_key_name).eq(partition_key_value)

    # Initialize the filter expression and expression attribute values
    filter_expression = None
    expression_attribute_values = {}

    # Build the filter expression based on provided parameters
    if min_rating is not None:
        filter_expression = Attr("rating").gte(min_rating)
        expression_attribute_values[":min_rating"] = min_rating

    if status_list and len(status_list) > 0:
        status_condition = None
        for i, status in enumerate(status_list):
            status_value_name = f":status{i}"
            expression_attribute_values[status_value_name] = status

            if status_condition is None:
                status_condition = Attr("status").eq(status)
            else:
                status_condition = status_condition | Attr("status").eq(status)

        if filter_expression is None:
            filter_expression = status_condition
        else:
            filter_expression = filter_expression & status_condition

    if max_price is not None:
```

```
        price_condition = Attr("price").lte(max_price)
        expression_attribute_values[":max_price"] = max_price

        if filter_expression is None:
            filter_expression = price_condition
        else:
            filter_expression = filter_expression & price_condition

    # Prepare the query parameters
    query_params = {"KeyConditionExpression": key_condition}

    if filter_expression:
        query_params["FilterExpression"] = filter_expression
        if expression_attribute_values:
            query_params["ExpressionAttributeValues"] =
 expression_attribute_values

    # Execute the query
    response = table.query(**query_params)
    return response


def query_with_complex_filter_and_or(
    table_name,
    partition_key_name,
    partition_key_value,
    category=None,
    min_rating=None,
    max_price=None,
):
    """
    Query a DynamoDB table with a complex filter expression using AND and OR
 operators.

    Args:
        table_name (str): The name of the DynamoDB table.
        partition_key_name (str): The name of the partition key attribute.
        partition_key_value (str): The value of the partition key to query.
        category (str, optional): Category value for filtering.
        min_rating (float, optional): Minimum rating value for filtering.
        max_price (float, optional): Maximum price value for filtering.

    Returns:
        dict: The response from DynamoDB containing the query results.
```

```
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # Start with the key condition expression
    key_condition = Key(partition_key_name).eq(partition_key_value)

    # Build a complex filter expression with AND and OR operators
    filter_expression = None
    expression_attribute_values = {}

    # Build the category condition
    if category:
        filter_expression = Attr("category").eq(category)
        expression_attribute_values[":category"] = category

    # Build the rating and price condition (rating >= min_rating OR price <=
max_price)
    rating_price_condition = None

    if min_rating is not None:
        rating_price_condition = Attr("rating").gte(min_rating)
        expression_attribute_values[":min_rating"] = min_rating

    if max_price is not None:
        price_condition = Attr("price").lte(max_price)
        expression_attribute_values[":max_price"] = max_price

        if rating_price_condition is None:
            rating_price_condition = price_condition
        else:
            rating_price_condition = rating_price_condition | price_condition

    # Combine the conditions
    if rating_price_condition:
        if filter_expression is None:
            filter_expression = rating_price_condition
        else:
            filter_expression = filter_expression & rating_price_condition

    # Prepare the query parameters
    query_params = {"KeyConditionExpression": key_condition}
```

```
    if filter_expression:
        query_params["FilterExpression"] = filter_expression
        if expression_attribute_values:
            query_params["ExpressionAttributeValues"] =
expression_attribute_values

    # Execute the query
    response = table.query(**query_params)
    return response
```

- For API details, see Query in *AWS SDK for Python (Boto3) API Reference*.

For a complete list of AWS SDK developer guides and code examples, see Using DynamoDB with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Query a DynamoDB table with a dynamic filter expression with an AWS SDK

The following code examples show how to query a table with a dynamic filter expression.

- Build filter expressions dynamically at runtime.
- Construct filter conditions based on user input or application state.
- Add or remove filter criteria conditionally.

Java

SDK for Java 2.x

Query a DynamoDB table with a dynamically constructed filter expression using AWS SDK for Java 2.x.

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.QueryRequest;
```

```
import software.amazon.awssdk.services.dynamodb.model.QueryResponse;
import software.amazon.awssdk.services.dynamodb.model.ResourceNotFoundException;

import java.util.HashMap;
import java.util.Map;

    public static QueryResponse queryWithDynamicFilter(
        final String tableName,
        final String partitionKeyName,
        final String partitionKeyValue,
        final Map<String, Object> filterCriteria,
        final Region region,
        final DynamoDbClient dynamoDbClient) {

        validateParameters(tableName, partitionKeyName, partitionKeyValue,
 filterCriteria);

        DynamoDbClient ddbClient = dynamoDbClient;
        boolean shouldClose = false;

        try {
            if (ddbClient == null) {
                ddbClient = createClient(region);
                shouldClose = true;
            }

            final QueryWithDynamicFilter queryHelper = new
 QueryWithDynamicFilter(ddbClient);
            return queryHelper.queryWithDynamicFilter(tableName,
 partitionKeyName, partitionKeyValue, filterCriteria);
        } catch (ResourceNotFoundException e) {
            System.err.println("Table not found: " + tableName);
            throw e;
        } catch (DynamoDbException e) {
            System.err.println("Failed to execute dynamic filter query: " +
 e.getMessage());
            throw e;
        } catch (Exception e) {
            System.err.println("Unexpected error during query: " +
 e.getMessage());
            throw e;
        } finally {
            if (shouldClose && ddbClient != null) {
                ddbClient.close();
```

```
            }
        }
    }
```

Demonstrates how to use dynamic filter expressions with AWS SDK for Java 2.x.

```
    public static void main(String[] args) {
        final String usage =
            """
                Usage:
                    <tableName> <partitionKeyName> <partitionKeyValue>
<filterAttrName> <filterAttrValue> [region]
                Where:
                    tableName - The Amazon DynamoDB table to query.
                    partitionKeyName - The name of the partition key attribute.
                    partitionKeyValue - The value of the partition key to query.
                    filterAttrName - The name of the attribute to filter on.
                    filterAttrValue - The value to filter by.
                    region (optional) - The AWS region where the table exists.
(Default: us-east-1)
                """;

        if (args.length < 5) {
            System.out.println(usage);
            System.exit(1);
        }

        final String tableName = args[0];
        final String partitionKeyName = args[1];
        final String partitionKeyValue = args[2];
        final String filterAttrName = args[3];
        final String filterAttrValue = args[4];
        final Region region = args.length > 5 ? Region.of(args[5]) :
Region.US_EAST_1;

        System.out.println("Querying items with dynamic filter: " +
filterAttrName + " = " + filterAttrValue);

        try {
            // Using the builder pattern to create and execute the query
            final QueryResponse response = new DynamicFilterQueryBuilder()
                .withTableName(tableName)
```

```
                .withPartitionKeyName(partitionKeyName)
                .withPartitionKeyValue(partitionKeyValue)
                .withFilterCriterion(filterAttrName, filterAttrValue)
                .withRegion(region)
                .execute();

            // Process the results
            System.out.println("Found " + response.count() + " items:");
            response.items().forEach(item -> System.out.println(item));

            // Demonstrate multiple filter criteria
            System.out.println("\nNow querying with multiple filter criteria:");

            Map<String, Object> multipleFilters = new HashMap<>();
            multipleFilters.put(filterAttrName, filterAttrValue);
            multipleFilters.put("status", "active");

            final QueryResponse multiFilterResponse = new
DynamicFilterQueryBuilder()
                .withTableName(tableName)
                .withPartitionKeyName(partitionKeyName)
                .withPartitionKeyValue(partitionKeyValue)
                .withFilterCriteria(multipleFilters)
                .withRegion(region)
                .execute();

            System.out.println("Found " + multiFilterResponse.count() + " items
with multiple filters:");
            multiFilterResponse.items().forEach(item ->
System.out.println(item));

        } catch (IllegalArgumentException e) {
            System.err.println("Invalid input: " + e.getMessage());
            System.exit(1);
        } catch (ResourceNotFoundException e) {
            System.err.println("Table not found: " + tableName);
            System.exit(1);
        } catch (DynamoDbException e) {
            System.err.println("DynamoDB error: " + e.getMessage());
            System.exit(1);
        } catch (Exception e) {
            System.err.println("Unexpected error: " + e.getMessage());
            System.exit(1);
        }
```

```
    }
```

- For API details, see Query in *AWS SDK for Java 2.x API Reference*.

JavaScript

SDK for JavaScript (v3)

Query a DynamoDB table with a dynamically constructed filter expression using AWS SDK for JavaScript.

```
const { DynamoDBClient, QueryCommand } = require("@aws-sdk/client-dynamodb");

async function queryWithDynamicFilter(
  config,
  tableName,
  partitionKeyName,
  partitionKeyValue,
  sortKeyName,
  sortKeyValue,
  filterParams = {}
) {
  try {
    // Create DynamoDB client
    const client = new DynamoDBClient(config);

    // Initialize filter expression components
    let filterExpressions = [];
    const expressionAttributeValues = {
      ":pkValue": { S: partitionKeyValue },
      ":skValue": { S: sortKeyValue }
    };
    const expressionAttributeNames = {
      "#pk": partitionKeyName,
      "#sk": sortKeyName
    };

    // Add status filter if provided
    if (filterParams.status) {
      filterExpressions.push("status = :status");
      expressionAttributeValues[":status"] = { S: filterParams.status };
    }
```

```
    // Add minimum views filter if provided
    if (filterParams.minViews !== undefined) {
      filterExpressions.push("views >= :minViews");
      expressionAttributeValues[":minViews"] = { N:
 filterParams.minViews.toString() };
    }

    // Add author filter if provided
    if (filterParams.author) {
      filterExpressions.push("author = :author");
      expressionAttributeValues[":author"] = { S: filterParams.author };
    }

    // Construct the query input
    const input = {
      TableName: tableName,
      KeyConditionExpression: "#pk = :pkValue AND #sk = :skValue"
    };

    // Add filter expression if any filters were provided
    if (filterExpressions.length > 0) {
      input.FilterExpression = filterExpressions.join(" AND ");
    }

    // Add expression attribute names and values
    input.ExpressionAttributeNames = expressionAttributeNames;
    input.ExpressionAttributeValues = expressionAttributeValues;

    // Execute the query
    const command = new QueryCommand(input);
    return await client.send(command);
  } catch (error) {
    console.error(`Error querying with dynamic filter: ${error}`);
    throw error;
  }
}
```

- For API details, see Query in *AWS SDK for JavaScript API Reference*.

Python

SDK for Python (Boto3)

Query a DynamoDB table with a dynamically constructed filter expression using AWS SDK for Python (Boto3).

```
import boto3
from boto3.dynamodb.conditions import Attr, Key


def query_with_dynamic_filter(
    table_name, partition_key_name, partition_key_value, filter_conditions=None
):
    """
    Query a DynamoDB table with a dynamically constructed filter expression.

    Args:
        table_name (str): The name of the DynamoDB table.
        partition_key_name (str): The name of the partition key attribute.
        partition_key_value (str): The value of the partition key to query.
        filter_conditions (dict, optional): A dictionary of filter conditions
 where
            keys are attribute names and values are dictionaries with 'operator'
 and 'value'.
            Example: {'rating': {'operator': '>=', 'value': 4}, 'status':
 {'operator': '=', 'value': 'active'}}

    Returns:
        dict: The response from DynamoDB containing the query results.
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # Start with the key condition expression
    key_condition = Key(partition_key_name).eq(partition_key_value)

    # Initialize variables for the filter expression and attribute values
    filter_expression = None
    expression_attribute_values = {":pk_val": partition_key_value}

    # Dynamically build the filter expression if filter conditions are provided
    if filter_conditions:
```

```
        for attr_name, condition in filter_conditions.items():
            operator = condition.get("operator")
            value = condition.get("value")
            attr_value_name = f":{attr_name}"
            expression_attribute_values[attr_value_name] = value

            # Create the appropriate filter expression based on the operator
            current_condition = None
            if operator == "=":
                current_condition = Attr(attr_name).eq(value)
            elif operator == "!=":
                current_condition = Attr(attr_name).ne(value)
            elif operator == ">":
                current_condition = Attr(attr_name).gt(value)
            elif operator == ">=":
                current_condition = Attr(attr_name).gte(value)
            elif operator == "<":
                current_condition = Attr(attr_name).lt(value)
            elif operator == "<=":
                current_condition = Attr(attr_name).lte(value)
            elif operator == "contains":
                current_condition = Attr(attr_name).contains(value)
            elif operator == "begins_with":
                current_condition = Attr(attr_name).begins_with(value)

            # Combine with existing filter expression using AND
            if current_condition:
                if filter_expression is None:
                    filter_expression = current_condition
                else:
                    filter_expression = filter_expression & current_condition

    # Perform the query with the dynamically built filter expression
    query_params = {"KeyConditionExpression": key_condition}

    if filter_expression:
        query_params["FilterExpression"] = filter_expression

    response = table.query(**query_params)
    return response
```

Demonstrates how to use dynamic filter expressions with AWS SDK for Python (Boto3).

```
def example_usage():
    """Example of how to use the query_with_dynamic_filter function."""
    # Example parameters
    table_name = "Products"
    partition_key_name = "Category"
    partition_key_value = "Electronics"

    # Define dynamic filter conditions based on user input or runtime conditions
    user_min_rating = 4  # This could come from user input
    user_status_filter = "active"  # This could come from user input

    filter_conditions = {}

    # Only add conditions that are actually specified
    if user_min_rating is not None:
        filter_conditions["rating"] = {"operator": ">=", "value":
 user_min_rating}

    if user_status_filter:
        filter_conditions["status"] = {"operator": "=", "value":
 user_status_filter}

    print(
        f"Querying products in category '{partition_key_value}' with filter
 conditions: {filter_conditions}"
    )

    # Execute the query with dynamic filter
    response = query_with_dynamic_filter(
        table_name, partition_key_name, partition_key_value, filter_conditions
    )

    # Process the results
    items = response.get("Items", [])
    print(f"Found {len(items)} items")

    for item in items:
        print(f"Product: {item}")
```

- For API details, see Query in *AWS SDK for Python (Boto3) API Reference*.

For a complete list of AWS SDK developer guides and code examples, see Using DynamoDB with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Query a DynamoDB table with a filter expression and limit with an AWS SDK

The following code examples show how to query a table with a filter expression and limit.

- Apply filter expressions to query results with a limit on items evaluated.
- Understand how limit affects filtered query results.
- Control the maximum number of items processed in a query.

Java

SDK for Java 2.x

Query a DynamoDB table with a filter expression and limit using AWS SDK for Java 2.x.

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.QueryRequest;
import software.amazon.awssdk.services.dynamodb.model.QueryResponse;
import software.amazon.awssdk.services.dynamodb.model.ResourceNotFoundException;

import java.util.HashMap;
import java.util.Map;
import java.util.logging.Level;
import java.util.logging.Logger;

    public QueryResponse queryWithFilterAndLimit(
        final String tableName,
        final String partitionKeyName,
        final String partitionKeyValue,
        final String filterAttrName,
        final String filterAttrValue,
```

```
        final int limit) {

        CodeSampleUtils.validateTableParameters(tableName, partitionKeyName,
partitionKeyValue);
        CodeSampleUtils.validateStringParameter("Filter attribute name",
filterAttrName);
        CodeSampleUtils.validateStringParameter("Filter attribute value",
filterAttrValue);
        CodeSampleUtils.validatePositiveInteger("Limit", limit);

        // Create expression attribute names for the column names
        final Map<String, String> expressionAttributeNames = new HashMap<>();
        expressionAttributeNames.put(EXPRESSION_ATTRIBUTE_NAME_PK,
partitionKeyName);
        expressionAttributeNames.put(EXPRESSION_ATTRIBUTE_NAME_FILTER,
filterAttrName);

        // Create expression attribute values for the column values
        final Map<String, AttributeValue> expressionAttributeValues = new
HashMap<>();
        expressionAttributeValues.put(
            EXPRESSION_ATTRIBUTE_VALUE_PK,
            AttributeValue.builder().s(partitionKeyValue).build());
        expressionAttributeValues.put(
            EXPRESSION_ATTRIBUTE_VALUE_FILTER,
            AttributeValue.builder().s(filterAttrValue).build());

        // Create the filter expression
        final String filterExpression = "#filterAttr = :filterValue";

        // Create the query request
        final QueryRequest queryRequest = QueryRequest.builder()
            .tableName(tableName)
            .keyConditionExpression(KEY_CONDITION_EXPRESSION)
            .filterExpression(filterExpression)
            .expressionAttributeNames(expressionAttributeNames)
            .expressionAttributeValues(expressionAttributeValues)
            .limit(limit)
            .build();

        try {
            final QueryResponse response = dynamoDbClient.query(queryRequest);
            LOGGER.log(Level.INFO, "Query with filter and limit successful. Found
{0} items", response.count());
```

```
            LOGGER.log(
                Level.INFO, "ScannedCount: {0} (total items evaluated before
 filtering)", response.scannedCount());
            return response;
        } catch (ResourceNotFoundException e) {
            LOGGER.log(Level.SEVERE, "Table not found: {0}", tableName);
            throw e;
        } catch (DynamoDbException e) {
            LOGGER.log(Level.SEVERE, "Error querying with filter and limit: {0}",
 e.getMessage());
            throw e;
        }
    }
```

- For API details, see Query in *AWS SDK for Java 2.x API Reference*.

Python

SDK for Python (Boto3)

Query a DynamoDB table with a filter expression and limit using AWS SDK for Python (Boto3).

```
import boto3
from boto3.dynamodb.conditions import Attr, Key


def query_with_filter_and_limit(
    table_name,
    partition_key_name,
    partition_key_value,
    filter_attribute=None,
    filter_value=None,
    limit=10,
):
    """
    Query a DynamoDB table with a filter expression and limit the number of
 results.

    Args:
        table_name (str): The name of the DynamoDB table.
        partition_key_name (str): The name of the partition key attribute.
```

```
        partition_key_value (str): The value of the partition key to query.
        filter_attribute (str, optional): The attribute name to filter on.
        filter_value (any, optional): The value to compare against in the filter.
        limit (int, optional): The maximum number of items to evaluate. Defaults
 to 10.

    Returns:
        dict: The response from DynamoDB containing the query results.
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # Build the key condition expression
    key_condition = Key(partition_key_name).eq(partition_key_value)

    # Prepare the query parameters
    query_params = {"KeyConditionExpression": key_condition, "Limit": limit}

    # Add the filter expression if filter attributes are provided
    if filter_attribute and filter_value is not None:
        query_params["FilterExpression"] =
 Attr(filter_attribute).gt(filter_value)
        query_params["ExpressionAttributeValues"] = {":filter_value":
 filter_value}

    # Execute the query
    response = table.query(**query_params)
    return response
```

Demonstrates how to use filter expressions with limits in AWS SDK for Python (Boto3).

```
def example_usage():
    """Example of how to use the query_with_filter_and_limit function."""
    # Example parameters
    table_name = "ProductReviews"
    partition_key_name = "ProductId"
    partition_key_value = "P123456"
    filter_attribute = "Rating"
    filter_value = 3  # Filter for ratings > 3
```

```
    limit = 5

    print(f"Querying reviews for product '{partition_key_value}' with rating >
{filter_value}")
    print(f"Limiting to {limit} evaluated items")

    # Execute the query with filter and limit
    response = query_with_filter_and_limit(
        table_name, partition_key_name, partition_key_value, filter_attribute,
filter_value, limit
    )

    # Process the results
    items = response.get("Items", [])
    print(f"\nReturned {len(items)} items that passed the filter")

    for item in items:
        print(f"Review: {item}")

    # Explain the difference between Limit and actual results
    explain_limit_vs_results(response)

    # Check if there are more results
    if "LastEvaluatedKey" in response:
        print("\nThere are more results available. Use the LastEvaluatedKey for
pagination.")
    else:
        print("\nAll matching results have been retrieved.")
```

- For API details, see Query in *AWS SDK for Python (Boto3) API Reference*.

For a complete list of AWS SDK developer guides and code examples, see Using DynamoDB with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Query a DynamoDB table with nested attributes using an AWS SDK

The following code examples show how to query a table with nested attributes.

- Access and filter by nested attributes in DynamoDB items.
- Use document path expressions to reference nested elements.

Java

SDK for Java 2.x

Query a DynamoDB table with nested attributes using AWS SDK for Java 2.x.

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.QueryRequest;
import software.amazon.awssdk.services.dynamodb.model.QueryResponse;
import software.amazon.awssdk.services.dynamodb.model.ResourceNotFoundException;

import java.util.HashMap;
import java.util.Map;

    public QueryResponse queryWithNestedAttributes(
        final String tableName,
        final String partitionKeyName,
        final String partitionKeyValue,
        final String nestedPath,
        final String nestedAttr,
        final String nestedValue) {

        CodeSampleUtils.validateTableParameters(tableName, partitionKeyName,
 partitionKeyValue);
        CodeSampleUtils.validateStringParameter("Nested path", nestedPath);
        CodeSampleUtils.validateStringParameter("Nested attribute", nestedAttr);
        CodeSampleUtils.validateStringParameter("Nested value", nestedValue);

        // Split the nested path into components
        final String[] pathComponents = nestedPath.split("\\.");

        // Create expression attribute names for the column names
        final Map<String, String> expressionAttributeNames = new HashMap<>();
        expressionAttributeNames.put(EXPRESSION_ATTRIBUTE_NAME_PK,
 partitionKeyName);
```

```
        // Build the nested attribute reference using document path notation
        final StringBuilder nestedAttributeRef = new StringBuilder();
        for (int i = 0; i < pathComponents.length; i++) {
            final String aliasName = "#n" + i;
            expressionAttributeNames.put(aliasName, pathComponents[i]);

            if (i > 0) {
                nestedAttributeRef.append(".");
            }
            nestedAttributeRef.append(aliasName);
        }

        // Create expression attribute values for the column values
        final Map<String, AttributeValue> expressionAttributeValues = new
HashMap<>();
        expressionAttributeValues.put(
            EXPRESSION_ATTRIBUTE_VALUE_PK,
            AttributeValue.builder().s(partitionKeyValue).build());
        expressionAttributeValues.put(
            EXPRESSION_ATTRIBUTE_VALUE_NESTED,
            AttributeValue.builder().s(nestedValue).build());

        // Create the filter expression using the nested attribute reference
        final String filterExpression = nestedAttributeRef + " = :nestedValue";

        // Create the query request
        final QueryRequest queryRequest = QueryRequest.builder()
            .tableName(tableName)
            .keyConditionExpression(KEY_CONDITION_EXPRESSION)
            .filterExpression(filterExpression)
            .expressionAttributeNames(expressionAttributeNames)
            .expressionAttributeValues(expressionAttributeValues)
            .build();

        try {
            final QueryResponse response = dynamoDbClient.query(queryRequest);
            System.out.println("Query with nested attribute filter successful.
Found " + response.count() + " items");
            return response;
        } catch (ResourceNotFoundException e) {
            System.err.format("Error: The Amazon DynamoDB table \"%s\" can't be
found.\n", tableName);
            throw e;
        } catch (DynamoDbException e) {
```

```
            System.err.println("Error querying with nested attribute filter: " +
e.getMessage());
            throw e;
        }
    }
```

Demonstrates how to query a DynamoDB table with nested attributes.

```
    public static void main(String[] args) {
        final String usage =
            """
                Usage:
                    <tableName> <partitionKeyName> <partitionKeyValue>
<nestedPath> <nestedAttr> <nestedValue> [region]
                Where:
                    tableName - The Amazon DynamoDB table to query.
                    partitionKeyName - The name of the partition key attribute.
                    partitionKeyValue - The value of the partition key to query.
                    nestedPath - The path to the nested map attribute (e.g.,
"address").
                    nestedAttr - The name of the nested attribute (e.g., "city").
                    nestedValue - The value to filter by (e.g., "Seattle").
                    region (optional) - The AWS region where the table exists.
(Default: us-east-1)
                """;

        if (args.length < 6) {
            System.out.println(usage);
            System.exit(1);
        }

        final String tableName = args[0];
        final String partitionKeyName = args[1];
        final String partitionKeyValue = args[2];
        final String nestedPath = args[3];
        final String nestedAttr = args[4];
        final String nestedValue = args[5];
        final Region region = args.length > 6 ? Region.of(args[6]) :
Region.US_EAST_1;

        System.out.println("Querying items where " + partitionKeyName + " = " +
partitionKeyValue + " and " + nestedPath
```

```
            + "." + nestedAttr + " = " + nestedValue);

        try {
            // Using the builder pattern to create and execute the query
            final QueryResponse response = new NestedAttributeQueryBuilder()
                .withTableName(tableName)
                .withPartitionKeyName(partitionKeyName)
                .withPartitionKeyValue(partitionKeyValue)
                .withNestedPath(nestedPath)
                .withNestedAttribute(nestedAttr)
                .withNestedValue(nestedValue)
                .withRegion(region)
                .execute();

            // Process the results
            System.out.println("Found " + response.count() + " items:");
            response.items().forEach(item -> {
                System.out.println(item);

                // Extract and display the nested attribute for clarity
                if (item.containsKey(nestedPath) && item.get(nestedPath).hasM())
{
                    Map<String, AttributeValue> nestedMap =
item.get(nestedPath).m();
                    if (nestedMap.containsKey(nestedAttr)) {
                        System.out.println("  Nested attribute " + nestedPath +
"." + nestedAttr + ": "
                                + formatAttributeValue(nestedMap.get(nestedAttr)));
                    }
                }
            });

            System.out.println("\nNote: When working with nested attributes in
DynamoDB:");
            System.out.println("1. Use dot notation in filter expressions to
access nested attributes");
            System.out.println("2. Use expression attribute names for each
component of the path");
            System.out.println("3. Check if the nested attribute exists before
accessing it");

        } catch (IllegalArgumentException e) {
            System.err.println("Invalid input: " + e.getMessage());
            System.exit(1);
```

```
        } catch (ResourceNotFoundException e) {
            System.err.println("Table not found: " + tableName);
            System.exit(1);
        } catch (DynamoDbException e) {
            System.err.println("DynamoDB error: " + e.getMessage());
            System.exit(1);
        } catch (Exception e) {
            System.err.println("Unexpected error: " + e.getMessage());
            System.exit(1);
        }
    }
```

- For API details, see Query in *AWS SDK for Java 2.x API Reference*.

JavaScript

SDK for JavaScript (v3)

Query a DynamoDB table with nested attributes using AWS SDK for JavaScript.

```
const { DynamoDBClient, QueryCommand } = require("@aws-sdk/client-dynamodb");

/**
 * Queries a DynamoDB table filtering on a nested attribute
 *
 * @param {Object} config - AWS SDK configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {string} productId - The product ID to query by (partition key)
 * @param {string} category - The category to filter by (nested attribute)
 * @returns {Promise<Object>} - The query response
 */
async function queryWithNestedAttribute(
  config,
  tableName,
  productId,
  category
) {
  try {
    // Create DynamoDB client
    const client = new DynamoDBClient(config);

    // Construct the query input
```

```
    const input = {
      TableName: tableName,
      KeyConditionExpression: "product_id = :productId",
      FilterExpression: "details.category = :category",
      ExpressionAttributeValues: {
        ":productId": { S: productId },
        ":category": { S: category }
      }
    };

    // Execute the query
    const command = new QueryCommand(input);
    return await client.send(command);
  } catch (error) {
    console.error(`Error querying with nested attribute: ${error}`);
    throw error;
  }
}
```

- For API details, see Query in *AWS SDK for JavaScript API Reference*.

Python

SDK for Python (Boto3)

Query a DynamoDB table with nested attributes using AWS SDK for Python (Boto3).

```
from typing import Any, Dict, List

import boto3
from boto3.dynamodb.conditions import Attr, Key


def query_with_nested_attributes(
    table_name: str,
    partition_key_name: str,
    partition_key_value: str,
    nested_path: str,
    comparison_operator: str,
    comparison_value: Any,
) -> Dict[str, Any]:
    """
```

```
    Query a DynamoDB table and filter by nested attributes.

    Args:
        table_name (str): The name of the DynamoDB table.
        partition_key_name (str): The name of the partition key attribute.
        partition_key_value (str): The value of the partition key to query.
        nested_path (str): The path to the nested attribute (e.g.,
'specs.weight').
        comparison_operator (str): The comparison operator to use ('=', '!=',
'<', '<=', '>', '>=').
        comparison_value (any): The value to compare against.

    Returns:
        dict: The response from DynamoDB containing the query results.
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # Build the key condition expression
    key_condition = Key(partition_key_name).eq(partition_key_value)

    # Build the filter expression based on the nested attribute path and
comparison operator
    filter_expression = None
    if comparison_operator == "=":
        filter_expression = Attr(nested_path).eq(comparison_value)
    elif comparison_operator == "!=":
        filter_expression = Attr(nested_path).ne(comparison_value)
    elif comparison_operator == "<":
        filter_expression = Attr(nested_path).lt(comparison_value)
    elif comparison_operator == "<=":
        filter_expression = Attr(nested_path).lte(comparison_value)
    elif comparison_operator == ">":
        filter_expression = Attr(nested_path).gt(comparison_value)
    elif comparison_operator == ">=":
        filter_expression = Attr(nested_path).gte(comparison_value)
    elif comparison_operator == "contains":
        filter_expression = Attr(nested_path).contains(comparison_value)
    elif comparison_operator == "begins_with":
        filter_expression = Attr(nested_path).begins_with(comparison_value)

    # Execute the query with the filter expression
```

```
    response = table.query(KeyConditionExpression=key_condition,
 FilterExpression=filter_expression)

    return response


def query_with_multiple_nested_attributes(
    table_name: str,
    partition_key_name: str,
    partition_key_value: str,
    nested_conditions: List[Dict[str, Any]],
) -> Dict[str, Any]:
    """
    Query a DynamoDB table and filter by multiple nested attributes.

    Args:
        table_name (str): The name of the DynamoDB table.
        partition_key_name (str): The name of the partition key attribute.
        partition_key_value (str): The value of the partition key to query.
        nested_conditions (list): A list of dictionaries, each containing:
            - path (str): The path to the nested attribute
            - operator (str): The comparison operator
            - value (any): The value to compare against

    Returns:
        dict: The response from DynamoDB containing the query results.
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # Build the key condition expression
    key_condition = Key(partition_key_name).eq(partition_key_value)

    # Build the combined filter expression for all nested attributes
    combined_filter = None

    for condition in nested_conditions:
        if not isinstance(condition, dict):
            continue
        path = condition.get("path", "")
        operator = condition.get("operator", "")
        value = condition.get("value")
```

```
        if not path or not operator:
            continue

        # Build the individual filter expression
        current_filter = None
        if operator == "=":
            current_filter = Attr(path).eq(value)
        elif operator == "!=":
            current_filter = Attr(path).ne(value)
        elif operator == "<":
            current_filter = Attr(path).lt(value)
        elif operator == "<=":
            current_filter = Attr(path).lte(value)
        elif operator == ">":
            current_filter = Attr(path).gt(value)
        elif operator == ">=":
            current_filter = Attr(path).gte(value)
        elif operator == "contains":
            current_filter = Attr(path).contains(value)
        elif operator == "begins_with":
            current_filter = Attr(path).begins_with(value)

        # Combine with the existing filter using AND
        if current_filter:
            if combined_filter is None:
                combined_filter = current_filter
            else:
                combined_filter = combined_filter & current_filter

    # Execute the query with the combined filter expression
    response = table.query(KeyConditionExpression=key_condition,
FilterExpression=combined_filter)

    return response
```

- For API details, see Query in *AWS SDK for Python (Boto3) API Reference*.

For a complete list of AWS SDK developer guides and code examples, see Using DynamoDB with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Query a DynamoDB table with pagination using an AWS SDK

The following code examples show how to query a table with pagination.

- Implement pagination for DynamoDB query results.
- Use the LastEvaluatedKey to retrieve subsequent pages.
- Control the number of items per page with the Limit parameter.

Java

SDK for Java 2.x

Query a DynamoDB table with pagination using AWS SDK for Java 2.x.

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.QueryRequest;
import software.amazon.awssdk.services.dynamodb.model.QueryResponse;
import software.amazon.awssdk.services.dynamodb.model.ResourceNotFoundException;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

    public List<Map<String, AttributeValue>> queryWithPagination(
        final String tableName, final String partitionKeyName, final String
 partitionKeyValue, final int pageSize) {

        CodeSampleUtils.validateTableParameters(tableName, partitionKeyName,
 partitionKeyValue);
        CodeSampleUtils.validatePositiveInteger("Page size", pageSize);

        // Create expression attribute names for the column names
        final Map<String, String> expressionAttributeNames = new HashMap<>();
```

```
        expressionAttributeNames.put(EXPRESSION_ATTRIBUTE_NAME_PK,
partitionKeyName);

        // Create expression attribute values for the column values
        final Map<String, AttributeValue> expressionAttributeValues = new
HashMap<>();
        expressionAttributeValues.put(
            EXPRESSION_ATTRIBUTE_VALUE_PK,
            AttributeValue.builder().s(partitionKeyValue).build());

        // Create the query request
        QueryRequest.Builder queryRequestBuilder = QueryRequest.builder()
            .tableName(tableName)
            .keyConditionExpression(KEY_CONDITION_EXPRESSION)
            .expressionAttributeNames(expressionAttributeNames)
            .expressionAttributeValues(expressionAttributeValues)
            .limit(pageSize);

        // List to store all items from all pages
        final List<Map<String, AttributeValue>> allItems = new ArrayList<>();

        // Map to store the last evaluated key for pagination
        Map<String, AttributeValue> lastEvaluatedKey = null;
        int pageNumber = 1;

        try {
            do {
                // If we have a last evaluated key, use it for the next page
                if (lastEvaluatedKey != null) {
                    queryRequestBuilder.exclusiveStartKey(lastEvaluatedKey);
                }

                // Execute the query
                final QueryResponse response =
dynamoDbClient.query(queryRequestBuilder.build());

                // Process the current page of results
                final List<Map<String, AttributeValue>> pageItems =
response.items();
                allItems.addAll(pageItems);

                // Get the last evaluated key for the next page
                lastEvaluatedKey = response.lastEvaluatedKey();
                if (lastEvaluatedKey != null && lastEvaluatedKey.isEmpty()) {
```

```
                    lastEvaluatedKey = null;
                }

                System.out.println("Page " + pageNumber + ": Retrieved " +
pageItems.size() + " items (Running total: "
                    + allItems.size() + ")");

                pageNumber++;

            } while (lastEvaluatedKey != null);

            System.out.println("Query with pagination complete. Retrieved a total
of " + allItems.size()
                + " items across " + (pageNumber - 1) + " pages");

            return allItems;
        } catch (ResourceNotFoundException e) {
            System.err.format("Error: The Amazon DynamoDB table \"%s\" can't be
found.\n", tableName);
            throw e;
        } catch (DynamoDbException e) {
            System.err.println("Error querying with pagination: " +
e.getMessage());
            throw e;
        }
    }
```

Demonstrates how to query a DynamoDB table with pagination.

```
    public static void main(String[] args) {
        final String usage =
            """
                Usage:
                    <tableName> <partitionKeyName> <partitionKeyValue> [pageSize]
[region]
                Where:
                    tableName - The Amazon DynamoDB table to query.
                    partitionKeyName - The name of the partition key attribute.
                    partitionKeyValue - The value of the partition key to query.
                    pageSize (optional) - The maximum number of items to return
per page. (Default: 10)
```

```
                    region (optional) - The AWS region where the table exists.
(Default: us-east-1)
                """;

        if (args.length < 3) {
            System.out.println(usage);
            System.exit(1);
        }

        final String tableName = args[0];
        final String partitionKeyName = args[1];
        final String partitionKeyValue = args[2];
        final int pageSize = args.length > 3 ? Integer.parseInt(args[3]) : 10;
        final Region region = args.length > 4 ? Region.of(args[4]) :
Region.US_EAST_1;

        System.out.println("Querying items with pagination (page size: " +
pageSize + ")");

        try {
            // Using the builder pattern to create and execute the query
            final List<Map<String, AttributeValue>> allItems = new
PaginationQueryBuilder()
                .withTableName(tableName)
                .withPartitionKeyName(partitionKeyName)
                .withPartitionKeyValue(partitionKeyValue)
                .withPageSize(pageSize)
                .withRegion(region)
                .executeWithPagination();

            // Process the results
            System.out.println("\nSummary: Retrieved a total of " +
allItems.size() + " items");

            // Display the first few items as a sample
            final int sampleSize = Math.min(5, allItems.size());
            if (sampleSize > 0) {
                System.out.println("\nSample of retrieved items (first " +
sampleSize + "):");
                for (int i = 0; i < sampleSize; i++) {
                    System.out.println(allItems.get(i));
                }

                if (allItems.size() > sampleSize) {
```

```
                    System.out.println("... and " + (allItems.size() -
 sampleSize) + " more items");
                }
            }
        } catch (IllegalArgumentException e) {
            System.err.println("Invalid input: " + e.getMessage());
            System.exit(1);
        } catch (ResourceNotFoundException e) {
            System.err.println("Table not found: " + tableName);
            System.exit(1);
        } catch (DynamoDbException e) {
            System.err.println("DynamoDB error: " + e.getMessage());
            System.exit(1);
        } catch (Exception e) {
            System.err.println("Unexpected error: " + e.getMessage());
            System.exit(1);
        }
    }
```

- For API details, see Query in *AWS SDK for Java 2.x API Reference*.

JavaScript

SDK for JavaScript (v3)

Query a DynamoDB table with pagination using AWS SDK for JavaScript.

```
/**
 * Example demonstrating how to handle large query result sets in DynamoDB using
 pagination
 *
 * This example shows:
 * - How to use pagination to handle large result sets
 * - How to use LastEvaluatedKey to retrieve the next page of results
 * - How to construct subsequent query requests using ExclusiveStartKey
 */
const { DynamoDBClient, QueryCommand } = require("@aws-sdk/client-dynamodb");

/**
 * Queries a DynamoDB table with pagination to handle large result sets
 *
```

```
 * @param {Object} config - AWS SDK configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {string} partitionKeyName - The name of the partition key
 * @param {string} partitionKeyValue - The value of the partition key
 * @param {number} pageSize - Number of items per page
 * @returns {Promise<Array>} - All items from the query
 */
async function queryWithPagination(
  config,
  tableName,
  partitionKeyName,
  partitionKeyValue,
  pageSize = 25
) {
  try {
    // Create DynamoDB client
    const client = new DynamoDBClient(config);

    // Initialize variables for pagination
    let lastEvaluatedKey = undefined;
    const allItems = [];
    let pageCount = 0;

    // Loop until all pages are retrieved
    do {
      // Construct the query input
      const input = {
        TableName: tableName,
        KeyConditionExpression: "#pk = :pkValue",
        Limit: pageSize,
        ExpressionAttributeNames: {
          "#pk": partitionKeyName
        },
        ExpressionAttributeValues: {
          ":pkValue": { S: partitionKeyValue }
        }
      };

      // Add ExclusiveStartKey if we have a LastEvaluatedKey from a previous
 query
      if (lastEvaluatedKey) {
        input.ExclusiveStartKey = lastEvaluatedKey;
      }
```

```
      // Execute the query
      const command = new QueryCommand(input);
      const response = await client.send(command);

      // Process the current page of results
      pageCount++;
      console.log(`Processing page ${pageCount} with ${response.Items.length}
 items`);

      // Add the items from this page to our collection
      if (response.Items && response.Items.length > 0) {
        allItems.push(...response.Items);
      }

      // Get the LastEvaluatedKey for the next page
      lastEvaluatedKey = response.LastEvaluatedKey;

    } while (lastEvaluatedKey); // Continue until there are no more pages

    console.log(`Query complete. Retrieved ${allItems.length} items in
 ${pageCount} pages.`);
    return allItems;
  } catch (error) {
    console.error(`Error querying with pagination: ${error}`);
    throw error;
  }
}

/**
 * Example usage:
 *
 * // Query all items in the "AWS DynamoDB" forum with pagination
 * const allItems = await queryWithPagination(
 *   { region: "us-west-2" },
 *   "ForumThreads",
 *   "ForumName",
 *   "AWS DynamoDB",
 *   25 // 25 items per page
 * );
 *
 * console.log(`Total items retrieved: ${allItems.length}`);
 *
 * // Notes on pagination:
 * // - LastEvaluatedKey contains the primary key of the last evaluated item
```

```
 * // - When LastEvaluatedKey is undefined/null, there are no more items to
 retrieve
 * // - ExclusiveStartKey tells DynamoDB where to start the next page
 * // - Pagination helps manage memory usage for large result sets
 * // - Each page requires a separate network request to DynamoDB
 */

module.exports = { queryWithPagination };
```

- For API details, see Query in *AWS SDK for JavaScript API Reference*.

Python

SDK for Python (Boto3)

Query a DynamoDB table with pagination using AWS SDK for Python (Boto3).

```
import boto3
from boto3.dynamodb.conditions import Key


def query_with_pagination(
    table_name, partition_key_name, partition_key_value, page_size=25,
 max_pages=None
):
    """
    Query a DynamoDB table with pagination to handle large result sets.

    Args:
        table_name (str): The name of the DynamoDB table.
        partition_key_name (str): The name of the partition key attribute.
        partition_key_value (str): The value of the partition key to query.
        page_size (int, optional): The number of items to return per page.
 Defaults to 25.
        max_pages (int, optional): The maximum number of pages to retrieve. If
 None, retrieves all pages.

    Returns:
        list: All items retrieved from the query across all pages.
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
```

```
    table = dynamodb.Table(table_name)

    # Initialize variables for pagination
    last_evaluated_key = None
    page_count = 0
    all_items = []

    # Paginate through the results
    while True:
        # Check if we've reached the maximum number of pages
        if max_pages is not None and page_count >= max_pages:
            break

        # Prepare the query parameters
        query_params = {
            "KeyConditionExpression":
Key(partition_key_name).eq(partition_key_value),
            "Limit": page_size,
        }

        # Add the ExclusiveStartKey if we have a LastEvaluatedKey from a previous
query
        if last_evaluated_key:
            query_params["ExclusiveStartKey"] = last_evaluated_key

        # Execute the query
        response = table.query(**query_params)

        # Process the current page of results
        items = response.get("Items", [])
        all_items.extend(items)

        # Update pagination tracking
        page_count += 1

        # Get the LastEvaluatedKey for the next page, if any
        last_evaluated_key = response.get("LastEvaluatedKey")

        # If there's no LastEvaluatedKey, we've reached the end of the results
        if not last_evaluated_key:
            break

    return all_items
```

```
def query_with_pagination_generator(
    table_name, partition_key_name, partition_key_value, page_size=25
):
    """
    Query a DynamoDB table with pagination using a generator to handle large
 result sets.
    This approach is memory-efficient as it yields one page at a time.

    Args:
        table_name (str): The name of the DynamoDB table.
        partition_key_name (str): The name of the partition key attribute.
        partition_key_value (str): The value of the partition key to query.
        page_size (int, optional): The number of items to return per page.
 Defaults to 25.

    Yields:
        tuple: A tuple containing (items, page_number, last_page) where:
            - items is a list of items for the current page
            - page_number is the current page number (starting from 1)
            - last_page is a boolean indicating if this is the last page
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # Initialize variables for pagination
    last_evaluated_key = None
    page_number = 0

    # Paginate through the results
    while True:
        # Prepare the query parameters
        query_params = {
            "KeyConditionExpression":
 Key(partition_key_name).eq(partition_key_value),
            "Limit": page_size,
        }

        # Add the ExclusiveStartKey if we have a LastEvaluatedKey from a previous
 query
        if last_evaluated_key:
            query_params["ExclusiveStartKey"] = last_evaluated_key
```

```
        # Execute the query
        response = table.query(**query_params)

        # Get the current page of results
        items = response.get("Items", [])
        page_number += 1

        # Get the LastEvaluatedKey for the next page, if any
        last_evaluated_key = response.get("LastEvaluatedKey")

        # Determine if this is the last page
        is_last_page = last_evaluated_key is None

        # Yield the current page of results
        yield (items, page_number, is_last_page)

        # If there's no LastEvaluatedKey, we've reached the end of the results
        if is_last_page:
            break
```

- For API details, see Query in *AWS SDK for Python (Boto3) API Reference*.

For a complete list of AWS SDK developer guides and code examples, see Using DynamoDB with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Query a DynamoDB table with strongly consistent reads using an AWS SDK

The following code examples show how to query a table with strongly consistent reads.

- Configure the consistency level for DynamoDB queries.
- Use strongly consistent reads to get the most up-to-date data.
- Understand the tradeoffs between eventual consistency and strong consistency.

Java

SDK for Java 2.x

Query a DynamoDB table with configurable read consistency using AWS SDK for Java 2.x.

```
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.QueryRequest;
import software.amazon.awssdk.services.dynamodb.model.QueryResponse;
import software.amazon.awssdk.services.dynamodb.model.ResourceNotFoundException;

import java.util.HashMap;
import java.util.Map;
import java.util.logging.Level;
import java.util.logging.Logger;

    public QueryResponse queryWithConsistentReads(
        final String tableName,
        final String partitionKeyName,
        final String partitionKeyValue,
        final boolean useConsistentRead) {

        CodeSampleUtils.validateTableParameters(tableName, partitionKeyName,
 partitionKeyValue);

        // Create expression attribute names for the column names
        final Map<String, String> expressionAttributeNames = new HashMap<>();
        expressionAttributeNames.put(EXPRESSION_ATTRIBUTE_NAME_PK,
 partitionKeyName);

        // Create expression attribute values for the column values
        final Map<String, AttributeValue> expressionAttributeValues = new
 HashMap<>();
        expressionAttributeValues.put(
            EXPRESSION_ATTRIBUTE_VALUE_PK,
            AttributeValue.builder().s(partitionKeyValue).build());

        // Create the query request
        final QueryRequest queryRequest = QueryRequest.builder()
            .tableName(tableName)
            .keyConditionExpression(KEY_CONDITION_EXPRESSION)
            .expressionAttributeNames(expressionAttributeNames)
```

```
            .expressionAttributeValues(expressionAttributeValues)
            .consistentRead(useConsistentRead)
            .build();

        try {
            final QueryResponse response = dynamoDbClient.query(queryRequest);
            LOGGER.log(Level.INFO, "Query successful. Found {0} items",
 response.count());
            return response;
        } catch (ResourceNotFoundException e) {
            LOGGER.log(Level.SEVERE, "Table not found: {0}", tableName);
            throw e;
        } catch (DynamoDbException e) {
            LOGGER.log(Level.SEVERE, "Error querying with consistent reads", e);
            throw e;
        }
    }
```

- For API details, see Query in *AWS SDK for Java 2.x API Reference*.

JavaScript

SDK for JavaScript (v3)

Query a DynamoDB table with configurable read consistency using AWS SDK for JavaScript.

```
const { DynamoDBClient, QueryCommand } = require("@aws-sdk/client-dynamodb");

/**
 * Queries a DynamoDB table with configurable read consistency
 *
 * @param {Object} config - AWS SDK configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {string} partitionKeyName - The name of the partition key
 * @param {string} partitionKeyValue - The value of the partition key
 * @param {boolean} useConsistentRead - Whether to use strongly consistent reads
 * @returns {Promise<Object>} - The query response
 */
async function queryWithConsistentRead(
  config,
  tableName,
  partitionKeyName,
```

```
  partitionKeyValue,
  useConsistentRead = false
) {
  try {
    // Create DynamoDB client
    const client = new DynamoDBClient(config);

    // Construct the query input
    const input = {
      TableName: tableName,
      KeyConditionExpression: "#pk = :pkValue",
      ExpressionAttributeNames: {
        "#pk": partitionKeyName
      },
      ExpressionAttributeValues: {
        ":pkValue": { S: partitionKeyValue }
      },
      ConsistentRead: useConsistentRead
    };

    // Execute the query
    const command = new QueryCommand(input);
    return await client.send(command);
  } catch (error) {
    console.error(`Error querying with consistent read: ${error}`);
    throw error;
  }
}
```

- For API details, see Query in *AWS SDK for JavaScript API Reference*.

Python

SDK for Python (Boto3)

Query a DynamoDB table with the option for strongly consistent reads using AWS SDK for Python (Boto3).

```
import time

import boto3
from boto3.dynamodb.conditions import Key
```

```
def query_with_consistent_read(
    table_name,
    partition_key_name,
    partition_key_value,
    sort_key_name=None,
    sort_key_value=None,
    consistent_read=True,
):
    """
    Query a DynamoDB table with the option for strongly consistent reads.

    Args:
        table_name (str): The name of the DynamoDB table.
        partition_key_name (str): The name of the partition key attribute.
        partition_key_value (str): The value of the partition key to query.
        sort_key_name (str, optional): The name of the sort key attribute.
        sort_key_value (str, optional): The value of the sort key to query.
        consistent_read (bool, optional): Whether to use strongly consistent
 reads. Defaults to True.

    Returns:
        dict: The response from DynamoDB containing the query results.
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # Build the key condition expression
    key_condition = Key(partition_key_name).eq(partition_key_value)

    if sort_key_name and sort_key_value:
        key_condition = key_condition & Key(sort_key_name).eq(sort_key_value)

    # Perform the query with the consistent read option
    response = table.query(KeyConditionExpression=key_condition,
 ConsistentRead=consistent_read)

    return response
```

- For API details, see Query in *AWS SDK for Python (Boto3) API Reference*.

For a complete list of AWS SDK developer guides and code examples, see Using DynamoDB with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Query DynamoDB data using PartiQL SELECT statements with an AWS SDK

The following code example shows how to query data using PartiQL SELECT statements.

JavaScript

SDK for JavaScript (v3)

Query items from a DynamoDB table using PartiQL SELECT statements with AWS SDK for JavaScript.

```
/**
 * This example demonstrates how to query items from a DynamoDB table using
 PartiQL.
 * It shows different ways to select data with various index types.
 */
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
import {
  DynamoDBDocumentClient,
  ExecuteStatementCommand,
  BatchExecuteStatementCommand,
} from "@aws-sdk/lib-dynamodb";

/**
 * Select all items from a DynamoDB table using PartiQL.
 * Note: This should be used with caution on large tables.
 *
 * @param tableName - The name of the DynamoDB table
 * @returns The response from the ExecuteStatementCommand
 */
export const selectAllItems = async (tableName: string) => {
  const client = new DynamoDBClient({});
  const docClient = DynamoDBDocumentClient.from(client);
```

```
  const params = {
    Statement: `SELECT * FROM "${tableName}"`,
  };

  try {
    const data = await docClient.send(new ExecuteStatementCommand(params));
    console.log("Items retrieved successfully");
    return data;
  } catch (err) {
    console.error("Error retrieving items:", err);
    throw err;
  }
};

/**
 * Select an item by its primary key using PartiQL.
 *
 * @param tableName - The name of the DynamoDB table
 * @param partitionKeyName - The name of the partition key attribute
 * @param partitionKeyValue - The value of the partition key
 * @returns The response from the ExecuteStatementCommand
 */
export const selectItemByPartitionKey = async (
  tableName: string,
  partitionKeyName: string,
  partitionKeyValue: string | number
) => {
  const client = new DynamoDBClient({});
  const docClient = DynamoDBDocumentClient.from(client);

  const params = {
    Statement: `SELECT * FROM "${tableName}" WHERE ${partitionKeyName} = ?`,
    Parameters: [partitionKeyValue],
  };

  try {
    const data = await docClient.send(new ExecuteStatementCommand(params));
    console.log("Item retrieved successfully");
    return data;
  } catch (err) {
    console.error("Error retrieving item:", err);
    throw err;
  }
};
```

```
/**
 * Select an item by its composite key (partition key + sort key) using PartiQL.
 *
 * @param tableName - The name of the DynamoDB table
 * @param partitionKeyName - The name of the partition key attribute
 * @param partitionKeyValue - The value of the partition key
 * @param sortKeyName - The name of the sort key attribute
 * @param sortKeyValue - The value of the sort key
 * @returns The response from the ExecuteStatementCommand
 */
export const selectItemByCompositeKey = async (
  tableName: string,
  partitionKeyName: string,
  partitionKeyValue: string | number,
  sortKeyName: string,
  sortKeyValue: string | number
) => {
  const client = new DynamoDBClient({});
  const docClient = DynamoDBDocumentClient.from(client);

  const params = {
    Statement: `SELECT * FROM "${tableName}" WHERE ${partitionKeyName} = ? AND
 ${sortKeyName} = ?`,
    Parameters: [partitionKeyValue, sortKeyValue],
  };

  try {
    const data = await docClient.send(new ExecuteStatementCommand(params));
    console.log("Item retrieved successfully");
    return data;
  } catch (err) {
    console.error("Error retrieving item:", err);
    throw err;
  }
};

/**
 * Select items using a filter condition with PartiQL.
 *
 * @param tableName - The name of the DynamoDB table
 * @param filterAttribute - The attribute to filter on
 * @param filterValue - The value to filter by
 * @returns The response from the ExecuteStatementCommand
```

```
 */
export const selectItemsWithFilter = async (
  tableName: string,
  filterAttribute: string,
  filterValue: string | number
) => {
  const client = new DynamoDBClient({});
  const docClient = DynamoDBDocumentClient.from(client);

  const params = {
    Statement: `SELECT * FROM "${tableName}" WHERE ${filterAttribute} = ?`,
    Parameters: [filterValue],
  };

  try {
    const data = await docClient.send(new ExecuteStatementCommand(params));
    console.log("Items retrieved successfully");
    return data;
  } catch (err) {
    console.error("Error retrieving items:", err);
    throw err;
  }
};

/**
 * Select items using a begins_with function for prefix matching.
 * This is useful for querying hierarchical data.
 *
 * @param tableName - The name of the DynamoDB table
 * @param attributeName - The attribute to check for prefix
 * @param prefix - The prefix to match
 * @returns The response from the ExecuteStatementCommand
 */
export const selectItemsByPrefix = async (
  tableName: string,
  attributeName: string,
  prefix: string
) => {
  const client = new DynamoDBClient({});
  const docClient = DynamoDBDocumentClient.from(client);

  const params = {
    Statement: `SELECT * FROM "${tableName}" WHERE
 begins_with(${attributeName}, ?)`,
```

```
    Parameters: [prefix],
  };

  try {
    const data = await docClient.send(new ExecuteStatementCommand(params));
    console.log("Items retrieved successfully");
    return data;
  } catch (err) {
    console.error("Error retrieving items:", err);
    throw err;
  }
};

/**
 * Select items using a between condition for range queries.
 *
 * @param tableName - The name of the DynamoDB table
 * @param attributeName - The attribute to check for range
 * @param startValue - The start value of the range
 * @param endValue - The end value of the range
 * @returns The response from the ExecuteStatementCommand
 */
export const selectItemsByRange = async (
  tableName: string,
  attributeName: string,
  startValue: number | string,
  endValue: number | string
) => {
  const client = new DynamoDBClient({});
  const docClient = DynamoDBDocumentClient.from(client);

  const params = {
    Statement: `SELECT * FROM "${tableName}" WHERE ${attributeName} BETWEEN ?
 AND ?`,
    Parameters: [startValue, endValue],
  };

  try {
    const data = await docClient.send(new ExecuteStatementCommand(params));
    console.log("Items retrieved successfully");
    return data;
  } catch (err) {
    console.error("Error retrieving items:", err);
    throw err;
```

```
  }
};

/**
 * Example usage showing how to select items with different index types
 */
export const selectExamples = async () => {
  // Select all items from a table (use with caution on large tables)
  await selectAllItems("UsersTable");

  // Select by partition key (simple primary key)
  await selectItemByPartitionKey("UsersTable", "userId", "user123");

  // Select by composite key (partition key + sort key)
  await selectItemByCompositeKey("OrdersTable", "orderId", "order456",
 "productId", "prod789");

  // Select with a filter condition (can use any attribute)
  await selectItemsWithFilter("UsersTable", "userType", "premium");

  // Select items with a prefix (useful for hierarchical data)
  await selectItemsByPrefix("ProductsTable", "category", "electronics");

  // Select items within a range (useful for numeric or date ranges)
  await selectItemsByRange("OrdersTable", "orderDate", "2023-01-01",
 "2023-12-31");
};
```

- For API details, see the following topics in *AWS SDK for JavaScript API Reference*.
  - BatchExecuteStatement
  - ExecuteStatement

For a complete list of AWS SDK developer guides and code examples, see Using DynamoDB with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Query a DynamoDB table for TTL items using an AWS SDK

The following code examples show how to query for TTL items.

Java

SDK for Java 2.x

Query Filtered Expression to gather TTL items in a DynamoDB table using AWS SDK for Java 2.x.

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.QueryRequest;
import software.amazon.awssdk.services.dynamodb.model.QueryResponse;
import software.amazon.awssdk.services.dynamodb.model.ResourceNotFoundException;

import java.util.Map;
import java.util.Optional;

        final QueryRequest request = QueryRequest.builder()
            .tableName(tableName)
            .keyConditionExpression(KEY_CONDITION_EXPRESSION)
            .filterExpression(FILTER_EXPRESSION)
            .expressionAttributeNames(expressionAttributeNames)
            .expressionAttributeValues(expressionAttributeValues)
            .build();

        try (DynamoDbClient ddb = dynamoDbClient != null
            ? dynamoDbClient
            : DynamoDbClient.builder().region(region).build()) {
            final QueryResponse response = ddb.query(request);
            System.out.println("Query successful. Found " + response.count() + "
 items that have not expired yet.");

            // Print each item
            response.items().forEach(item -> {
                System.out.println("Item: " + item);
            });

            return 0;
        } catch (ResourceNotFoundException e) {
            System.err.format(TABLE_NOT_FOUND_ERROR, tableName);
            throw e;
        } catch (DynamoDbException e) {
            System.err.println(e.getMessage());
```

```
            throw e;
        }
```

- For API details, see Query in *AWS SDK for Java 2.x API Reference*.

JavaScript

SDK for JavaScript (v3)

Query Filtered Expression to gather TTL items in a DynamoDB table using AWS SDK for JavaScript.

```
import { DynamoDBClient, QueryCommand } from "@aws-sdk/client-dynamodb";
import { marshall, unmarshall } from "@aws-sdk/util-dynamodb";

export const queryFiltered = async (tableName, primaryKey, region = 'us-east-1')
 => {
    const client = new DynamoDBClient({
        region: region,
        endpoint: `https://dynamodb.${region}.amazonaws.com`
    });

    const currentTime = Math.floor(Date.now() / 1000);

    const params = {
        TableName: tableName,
        KeyConditionExpression: "#pk = :pk",
        FilterExpression: "#ea > :ea",
        ExpressionAttributeNames: {
            "#pk": "primaryKey",
            "#ea": "expireAt"
        },
        ExpressionAttributeValues: marshall({
            ":pk": primaryKey,
            ":ea": currentTime
        })
    };

    try {
        const { Items } = await client.send(new QueryCommand(params));
        Items.forEach(item => {
            console.log(unmarshall(item))
```

```
        });
        return Items;
    } catch (err) {
        console.error(`Error querying items: ${err}`);
        throw err;
    }
}

// Example usage (commented out for testing)
// queryFiltered('your-table-name', 'your-partition-key-value');
```

- For API details, see Query in *AWS SDK for JavaScript API Reference*.

Python

SDK for Python (Boto3)

Query Filtered Expression to gather TTL items in a DynamoDB table using AWS SDK for Python (Boto3).

```
from datetime import datetime

import boto3


def query_dynamodb_items(table_name, partition_key):
    """

    :param table_name: Name of the DynamoDB table
    :param partition_key:
    :return:
    """
    try:
        # Initialize a DynamoDB resource
        dynamodb = boto3.resource("dynamodb", region_name="us-east-1")

        # Specify your table
        table = dynamodb.Table(table_name)

        # Get the current time in epoch format
        current_time = int(datetime.now().timestamp())
```

```
        # Perform the query operation with a filter expression to exclude expired
 items
        # response = table.query(
        #
 KeyConditionExpression=boto3.dynamodb.conditions.Key('partitionKey').eq(partition_key),
        #
 FilterExpression=boto3.dynamodb.conditions.Attr('expireAt').gt(current_time)
        # )
        response = table.query(

 KeyConditionExpression=dynamodb.conditions.Key("partitionKey").eq(partition_key),

 FilterExpression=dynamodb.conditions.Attr("expireAt").gt(current_time),
        )

        # Print the items that are not expired
        for item in response["Items"]:
            print(item)

    except Exception as e:
        print(f"Error querying items: {e}")


# Call the function with your values
query_dynamodb_items("Music", "your-partition-key-value")
```

- For API details, see Query in *AWS SDK for Python (Boto3) API Reference*.

For a complete list of AWS SDK developer guides and code examples, see Using DynamoDB with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Query DynamoDB tables using date and time patterns with an AWS SDK

The following code examples show how to query tables using date and time patterns.

- Store and query date/time values in DynamoDB.
- Implement date range queries using sort keys.
- Format date strings for effective querying.

Java

SDK for Java 2.x

Query using date ranges in sort keys with AWS SDK for Java 2.x.

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.QueryRequest;
import software.amazon.awssdk.services.dynamodb.model.QueryResponse;
import software.amazon.awssdk.services.dynamodb.model.ResourceNotFoundException;

import java.time.LocalDate;
import java.util.HashMap;
import java.util.Map;
import java.util.logging.Level;
import java.util.logging.Logger;

    public QueryResponse queryWithDateRange(
        final String tableName,
        final String partitionKeyName,
        final String partitionKeyValue,
        final String dateKeyName,
        final LocalDate startDate,
        final LocalDate endDate) {

        // Focus on query logic, assuming parameters are valid
        if (startDate == null || endDate == null) {
            throw new IllegalArgumentException("Start date and end date cannot be
 null");
        }

        if (endDate.isBefore(startDate)) {
            throw new IllegalArgumentException("End date must be after start
 date");
        }

        // Format dates as ISO strings for DynamoDB (using just the date part)
        final String formattedStartDate = startDate.toString();
        final String formattedEndDate = endDate.toString();

        // Create expression attribute names for the column names
```

```
        final Map<String, String> expressionAttributeNames = new HashMap<>();
        expressionAttributeNames.put(EXPRESSION_ATTRIBUTE_NAME_PK,
partitionKeyName);
        expressionAttributeNames.put(EXPRESSION_ATTRIBUTE_NAME_SK, dateKeyName);

        // Create expression attribute values for the column values
        final Map<String, AttributeValue> expressionAttributeValues = new
HashMap<>();
        expressionAttributeValues.put(
            EXPRESSION_ATTRIBUTE_VALUE_PK,
            AttributeValue.builder().s(partitionKeyValue).build());
        expressionAttributeValues.put(
            EXPRESSION_ATTRIBUTE_VALUE_START_DATE,
            AttributeValue.builder().s(formattedStartDate).build());
        expressionAttributeValues.put(
            EXPRESSION_ATTRIBUTE_VALUE_END_DATE,
            AttributeValue.builder().s(formattedEndDate).build());

        // Create the query request
        final QueryRequest queryRequest = QueryRequest.builder()
            .tableName(tableName)
            .keyConditionExpression(KEY_CONDITION_EXPRESSION)
            .expressionAttributeNames(expressionAttributeNames)
            .expressionAttributeValues(expressionAttributeValues)
            .build();

        try {
            final QueryResponse response = dynamoDbClient.query(queryRequest);
            LOGGER.log(Level.INFO, "Query by date range successful. Found {0}
items", response.count());
            return response;
        } catch (ResourceNotFoundException e) {
            LOGGER.log(Level.SEVERE, "Table not found: {0}", tableName);
            throw e;
        } catch (DynamoDbException e) {
            LOGGER.log(Level.SEVERE, "Error querying by date range: {0}",
e.getMessage());
            throw e;
        }
    }
```

Query using date-time variables with AWS SDK for Java 2.x.

```
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.QueryRequest;
import software.amazon.awssdk.services.dynamodb.model.QueryResponse;
import software.amazon.awssdk.services.dynamodb.model.ResourceNotFoundException;

import java.time.Instant;
import java.time.LocalDateTime;
import java.time.ZoneOffset;
import java.util.HashMap;
import java.util.Map;

    public QueryResponse queryWithDateTime(
        final String tableName,
        final String partitionKeyName,
        final String partitionKeyValue,
        final String dateKeyName,
        final String startDate,
        final String endDate) {

        CodeSampleUtils.validateTableParameters(tableName, partitionKeyName,
 partitionKeyValue);
        CodeSampleUtils.validateDateRangeParameters(dateKeyName, startDate,
 endDate);
        CodeSampleUtils.validateDateFormat("Start date", startDate);
        CodeSampleUtils.validateDateFormat("End date", endDate);

        // Create expression attribute names for the column names
        final Map<String, String> expressionAttributeNames = new HashMap<>();
        expressionAttributeNames.put(EXPRESSION_ATTRIBUTE_NAME_PK,
 partitionKeyName);
        expressionAttributeNames.put("#dateKey", dateKeyName);

        // Create expression attribute values for the column values
        final Map<String, AttributeValue> expressionAttributeValues = new
 HashMap<>();
        expressionAttributeValues.put(
            EXPRESSION_ATTRIBUTE_VALUE_PK,
            AttributeValue.builder().s(partitionKeyValue).build());
        expressionAttributeValues.put(
            ":startDate", AttributeValue.builder().s(startDate).build());
        expressionAttributeValues.put(
```

```
            ":endDate", AttributeValue.builder().s(endDate).build());

        // Create the query request
        final QueryRequest queryRequest = QueryRequest.builder()
            .tableName(tableName)
            .keyConditionExpression(KEY_CONDITION_EXPRESSION)
            .expressionAttributeNames(expressionAttributeNames)
            .expressionAttributeValues(expressionAttributeValues)
            .build();

        try {
            final QueryResponse response = dynamoDbClient.query(queryRequest);
            System.out.println("Query successful. Found " + response.count() + "
items");
            return response;
        } catch (ResourceNotFoundException e) {
            System.err.format("Error: The Amazon DynamoDB table \"%s\" can't be
found.\n", tableName);
            throw e;
        } catch (DynamoDbException e) {
            System.err.println("Error querying with date range: " +
e.getMessage());
            throw e;
        }
    }
```

Query within date ranges in Unix epoch timestamps with AWS SDK for Java 2.x.

```
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.QueryRequest;
import software.amazon.awssdk.services.dynamodb.model.QueryResponse;
import software.amazon.awssdk.services.dynamodb.model.ResourceNotFoundException;

import java.time.Instant;
import java.time.LocalDateTime;
import java.time.ZoneOffset;
import java.util.HashMap;
import java.util.Map;

    public QueryResponse queryWithDateTimeEpoch(
```

```
        final String tableName,
        final String partitionKeyName,
        final String partitionKeyValue,
        final String dateKeyName,
        final long startEpoch,
        final long endEpoch) {

        CodeSampleUtils.validateTableParameters(tableName, partitionKeyName,
partitionKeyValue);
        CodeSampleUtils.validateStringParameter("Date key name", dateKeyName);
        CodeSampleUtils.validateEpochTimestamp("Start epoch", startEpoch);
        CodeSampleUtils.validateEpochTimestamp("End epoch", endEpoch);

        // Create expression attribute names for the column names
        final Map<String, String> expressionAttributeNames = new HashMap<>();
        expressionAttributeNames.put(EXPRESSION_ATTRIBUTE_NAME_PK,
partitionKeyName);
        expressionAttributeNames.put("#dateKey", dateKeyName);

        // Create expression attribute values for the column values
        final Map<String, AttributeValue> expressionAttributeValues = new
HashMap<>();
        expressionAttributeValues.put(
            EXPRESSION_ATTRIBUTE_VALUE_PK,
            AttributeValue.builder().s(partitionKeyValue).build());
        expressionAttributeValues.put(
            ":startDate",
AttributeValue.builder().n(String.valueOf(startEpoch)).build());
        expressionAttributeValues.put(
            ":endDate",
AttributeValue.builder().n(String.valueOf(endEpoch)).build());

        // Create the query request
        final QueryRequest queryRequest = QueryRequest.builder()
            .tableName(tableName)
            .keyConditionExpression(KEY_CONDITION_EXPRESSION)
            .expressionAttributeNames(expressionAttributeNames)
            .expressionAttributeValues(expressionAttributeValues)
            .build();

        try {
            final QueryResponse response = dynamoDbClient.query(queryRequest);
            System.out.println("Query successful. Found " + response.count() + "
items");
```

```
            return response;
        } catch (ResourceNotFoundException e) {
            System.err.format("Error: The Amazon DynamoDB table \"%s\" can't be
 found.\n", tableName);
            throw e;
        } catch (DynamoDbException e) {
            System.err.println("Error querying with epoch timestamps: " +
 e.getMessage());
            throw e;
        }
    }
```

Query within date ranges using LocalDateTime objects with AWS SDK for Java 2.x.

```
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.QueryRequest;
import software.amazon.awssdk.services.dynamodb.model.QueryResponse;
import software.amazon.awssdk.services.dynamodb.model.ResourceNotFoundException;

import java.time.Instant;
import java.time.LocalDateTime;
import java.time.ZoneOffset;
import java.util.HashMap;
import java.util.Map;

    public QueryResponse queryWithDateTimeLocalDateTime(
        final String tableName,
        final String partitionKeyName,
        final String partitionKeyValue,
        final String dateKeyName,
        final LocalDateTime startDateTime,
        final LocalDateTime endDateTime) {

        CodeSampleUtils.validateTableParameters(tableName, partitionKeyName,
 partitionKeyValue);
        CodeSampleUtils.validateStringParameter("Date key name", dateKeyName);
        if (startDateTime == null || endDateTime == null) {
            throw new IllegalArgumentException("Start and end LocalDateTime must
 not be null");
        }
```

```
        // Convert LocalDateTime to ISO-8601 strings in UTC with the correct
 format
        final String startDate =
 startDateTime.atZone(ZoneOffset.UTC).format(DATE_TIME_FORMATTER);
        final String endDate =
 endDateTime.atZone(ZoneOffset.UTC).format(DATE_TIME_FORMATTER);

        return queryWithDateTime(tableName, partitionKeyName, partitionKeyValue,
 dateKeyName, startDate, endDate);
    }
```

- For API details, see Query in *AWS SDK for Java 2.x API Reference*.

JavaScript

SDK for JavaScript (v3)

Query using date ranges in sort keys with AWS SDK for JavaScript.

```
const { DynamoDBClient, QueryCommand } = require("@aws-sdk/client-dynamodb");

/**
 * Queries a DynamoDB table for items within a specific date range on the sort
 key
 *
 * @param {Object} config - AWS SDK configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {string} partitionKeyName - The name of the partition key
 * @param {string} partitionKeyValue - The value of the partition key
 * @param {string} sortKeyName - The name of the sort key (must be a date/time
 attribute)
 * @param {Date} startDate - The start date for the range query
 * @param {Date} endDate - The end date for the range query
 * @returns {Promise<Object>} - The query response
 */
async function queryByDateRangeOnSortKey(
  config,
  tableName,
  partitionKeyName,
  partitionKeyValue,
  sortKeyName,
```

```
  startDate,
  endDate
) {
  try {
    // Create DynamoDB client
    const client = new DynamoDBClient(config);

    // Format dates as ISO strings for DynamoDB
    const formattedStartDate = startDate.toISOString();
    const formattedEndDate = endDate.toISOString();

    // Construct the query input
    const input = {
      TableName: tableName,
      KeyConditionExpression: '#pk = :pkValue AND #sk BETWEEN :startDate
 AND :endDate',
      ExpressionAttributeNames: {
        "#pk": partitionKeyName,
        "#sk": sortKeyName
      },
      ExpressionAttributeValues: {
        ":pkValue": { S: partitionKeyValue },
        ":startDate": { S: formattedStartDate },
        ":endDate": { S: formattedEndDate }
      }
    };

    // Execute the query
    const command = new QueryCommand(input);
    return await client.send(command);
  } catch (error) {
    console.error(`Error querying by date range on sort key: ${error}`);
    throw error;
  }
}
```

Query using date-time variables with AWS SDK for JavaScript.

```
const { DynamoDBClient, QueryCommand } = require("@aws-sdk/client-dynamodb");

/**
 * Queries a DynamoDB table for items within a specific date range
```

```
 *
 * @param {Object} config - AWS SDK configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {string} partitionKeyName - The name of the partition key
 * @param {string} partitionKeyValue - The value of the partition key
 * @param {string} dateKeyName - The name of the date attribute to filter on
 * @param {Date} startDate - The start date for the range query
 * @param {Date} endDate - The end date for the range query
 * @returns {Promise<Object>} - The query response
 */
async function queryByDateRange(
  config,
  tableName,
  partitionKeyName,
  partitionKeyValue,
  dateKeyName,
  startDate,
  endDate
) {
  try {
    // Create DynamoDB client
    const client = new DynamoDBClient(config);

    // Format dates as ISO strings for DynamoDB
    const formattedStartDate = startDate.toISOString();
    const formattedEndDate = endDate.toISOString();

    // Construct the query input
    const input = {
      TableName: tableName,
      KeyConditionExpression: `#pk = :pkValue AND #dateAttr BETWEEN :startDate
 AND :endDate`,
      ExpressionAttributeNames: {
        "#pk": partitionKeyName,
        "#dateAttr": dateKeyName
      },
      ExpressionAttributeValues: {
        ":pkValue": { S: partitionKeyValue },
        ":startDate": { S: formattedStartDate },
        ":endDate": { S: formattedEndDate }
      }
    };

    // Execute the query
```

```
    const command = new QueryCommand(input);
    return await client.send(command);
  } catch (error) {
    console.error(`Error querying by date range: ${error}`);
    throw error;
  }
}
```

- For API details, see Query in *AWS SDK for JavaScript API Reference*.

Python

SDK for Python (Boto3)

Query using date ranges in sort keys with AWS SDK for Python (Boto3).

```
from datetime import datetime, timedelta

import boto3
from boto3.dynamodb.conditions import Key


def query_with_date_range(
    table_name, partition_key_name, partition_key_value, sort_key_name,
 start_date, end_date
):
    """
    Query a DynamoDB table with a date range on the sort key.

    Args:
        table_name (str): The name of the DynamoDB table.
        partition_key_name (str): The name of the partition key attribute.
        partition_key_value (str): The value of the partition key to query.
        sort_key_name (str): The name of the sort key attribute (containing date
 values).
        start_date (datetime): The start date for the query range.
        end_date (datetime): The end date for the query range.

    Returns:
        dict: The response from DynamoDB containing the query results.
    """
    # Initialize the DynamoDB resource
```

```
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # Format the date values as ISO 8601 strings
    # DynamoDB works well with ISO format for date values
    start_date_str = start_date.isoformat()
    end_date_str = end_date.isoformat()

    # Perform the query with a date range on the sort key using BETWEEN operator
    key_condition = Key(partition_key_name).eq(partition_key_value) &
 Key(sort_key_name).between(
        start_date_str, end_date_str
    )

    response = table.query(
        KeyConditionExpression=key_condition,
        ExpressionAttributeValues={
            ":pk_val": partition_key_value,
            ":start_date": start_date_str,
            ":end_date": end_date_str,
        },
    )

    return response


def query_with_date_range_by_month(
    table_name, partition_key_name, partition_key_value, sort_key_name, year,
 month
):
    """
    Query a DynamoDB table for a specific month's data.

    Args:
        table_name (str): The name of the DynamoDB table.
        partition_key_name (str): The name of the partition key attribute.
        partition_key_value (str): The value of the partition key to query.
        sort_key_name (str): The name of the sort key attribute (containing date
 values).
        year (int): The year to query.
        month (int): The month to query (1-12).

    Returns:
        dict: The response from DynamoDB containing the query results.
```

```
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # Calculate the start and end dates for the specified month
    if month == 12:
        next_year = year + 1
        next_month = 1
    else:
        next_year = year
        next_month = month + 1

    start_date = datetime(year, month, 1)
    end_date = datetime(next_year, next_month, 1) - timedelta(microseconds=1)

    # Format the date values as ISO 8601 strings
    start_date_str = start_date.isoformat()
    end_date_str = end_date.isoformat()

    # Perform the query with a date range on the sort key
    key_condition = Key(partition_key_name).eq(partition_key_value) &
 Key(sort_key_name).between(
        start_date_str, end_date_str
    )

    response = table.query(KeyConditionExpression=key_condition)

    return response
```

Query using date-time variables with AWS SDK for Python (Boto3).

```
from datetime import datetime, timedelta

import boto3
from boto3.dynamodb.conditions import Key


def query_with_datetime(
```

```
    table_name, partition_key_name, partition_key_value, sort_key_name,
 start_date, end_date
):
    """
    Query a DynamoDB table with a date range filter on the sort key.

    Args:
        table_name (str): The name of the DynamoDB table.
        partition_key_name (str): The name of the partition key attribute.
        partition_key_value (str): The value of the partition key to query.
        sort_key_name (str): The name of the sort key attribute (containing date/
time values).
        start_date (datetime): The start date/time for the query range.
        end_date (datetime): The end date/time for the query range.

    Returns:
        dict: The response from DynamoDB containing the query results.
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # Format the date/time values as ISO 8601 strings
    # DynamoDB works well with ISO format for date/time values
    start_date_str = start_date.isoformat()
    end_date_str = end_date.isoformat()

    # Perform the query with a date range on the sort key
    key_condition = Key(partition_key_name).eq(partition_key_value) &
 Key(sort_key_name).between(
        start_date_str, end_date_str
    )

    response = table.query(
        KeyConditionExpression=key_condition,
        ExpressionAttributeValues={
            ":pk_val": partition_key_value,
            ":start_date": start_date_str,
            ":end_date": end_date_str,
        },
    )

    return response
```

```
def example_usage():
    """Example of how to use the query_with_datetime function."""
    # Example parameters
    table_name = "Events"
    partition_key_name = "EventType"
    partition_key_value = "UserLogin"
    sort_key_name = "Timestamp"

    # Create date/time variables for the query
    end_date = datetime.now()
    start_date = end_date - timedelta(days=7)  # Query events from the last 7
 days

    print(f"Querying events from {start_date.isoformat()} to
 {end_date.isoformat()}")

    # Execute the query
    response = query_with_datetime(
        table_name, partition_key_name, partition_key_value, sort_key_name,
 start_date, end_date
    )

    # Process the results
    items = response.get("Items", [])
    print(f"Found {len(items)} items")

    for item in items:
        print(f"Event: {item}")
```

- For API details, see Query in *AWS SDK for Python (Boto3) API Reference*.

For a complete list of AWS SDK developer guides and code examples, see Using DynamoDB with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Save EXIF and other image information using an AWS SDK

The following code example shows how to:

- Get EXIF information from a a JPG, JPEG, or PNG file.
- Upload the image file to an Amazon S3 bucket.
- Use Amazon Rekognition to identify the three top attributes (labels) in the file.
- Add the EXIF and label information to an Amazon DynamoDB table in the Region.

Rust

SDK for Rust

Get EXIF information from a JPG, JPEG, or PNG file, upload the image file to an Amazon S3 bucket, use Amazon Rekognition to identify the three top attributes (*labels* in Amazon Rekognition) in the file, and add the EXIF and label information to a Amazon DynamoDB table in the Region.

For complete source code and instructions on how to set up and run, see the full example on GitHub.

Services used in this example

- DynamoDB
- Amazon Rekognition
- Amazon S3

For a complete list of AWS SDK developer guides and code examples, see Using DynamoDB with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Set up Attribute-Based Access Control for DynamoDB using AWS Command Line Interface v2

The following code example shows how to implement Attribute-Based Access Control (ABAC) for DynamoDB.

- Create an IAM policy for ABAC.
- Create tables with tags for different departments.
- List and filter tables based on tags.

Bash

AWS CLI with Bash script

Create an IAM policy for ABAC.

```
# Step 1: Create a policy document for ABAC
cat > abac-policy.json << 'EOF'
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "dynamodb:GetItem",
        "dynamodb:BatchGetItem",
        "dynamodb:Query",
        "dynamodb:Scan"
      ],
      "Resource": "arn:aws:dynamodb:*:*:table/*",
      "Condition": {
        "StringEquals": {
          "aws:ResourceTag/Department": "${aws:PrincipalTag/Department}"
        }
      }
    },
    {
      "Effect": "Allow",
      "Action": [
        "dynamodb:PutItem",
        "dynamodb:UpdateItem",
        "dynamodb:DeleteItem",
        "dynamodb:BatchWriteItem"
      ],
      "Resource": "arn:aws:dynamodb:*:*:table/*",
      "Condition": {
        "StringEquals": {
          "aws:ResourceTag/Department": "${aws:PrincipalTag/Department}",
```

```
          "aws:ResourceTag/Environment": "Development"
        }
      }
    }
  ]
}
EOF

# Step 2: Create the IAM policy
aws iam create-policy \
    --policy-name DynamoDBDepartmentBasedAccess \
    --policy-document file://abac-policy.json
```

Create tables with tags for different departments.

```
# Create a DynamoDB table with tags for ABAC
aws dynamodb create-table \
    --table-name FinanceData \
    --attribute-definitions \
        AttributeName=RecordID,AttributeType=S \
    --key-schema \
        AttributeName=RecordID,KeyType=HASH \
    --billing-mode PAY_PER_REQUEST \
    --tags \
        Key=Department,Value=Finance \
        Key=Environment,Value=Development

# Create another table with different tags
aws dynamodb create-table \
    --table-name MarketingData \
    --attribute-definitions \
        AttributeName=RecordID,AttributeType=S \
    --key-schema \
        AttributeName=RecordID,KeyType=HASH \
    --billing-mode PAY_PER_REQUEST \
    --tags \
        Key=Department,Value=Marketing \
        Key=Environment,Value=Production
```

List and filter tables based on tags.

```
# List all DynamoDB tables
echo "Listing all tables:"
aws dynamodb list-tables

# Get ARNs for all tables
echo -e "\nGetting ARNs for all tables:"
TABLE_ARNS=$(aws dynamodb list-tables --query "TableNames[*]" --output text |
 xargs -I {} aws dynamodb describe-table --table-name {} --query "Table.TableArn"
 --output text)

# For each table ARN, list its tags
echo -e "\nListing tags for each table:"
for ARN in $TABLE_ARNS; do
    TABLE_NAME=$(echo $ARN | awk -F/ '{print $2}')
    echo -e "\nTags for table: $TABLE_NAME"
    aws dynamodb list-tags-of-resource --resource-arn $ARN
done

# Example: Find tables with a specific tag
echo -e "\nFinding tables with Environment=Production tag:"
for ARN in $TABLE_ARNS; do
    TABLE_NAME=$(echo $ARN | awk -F/ '{print $2}')
    TAGS=$(aws dynamodb list-tags-of-resource --resource-arn $ARN --query "Tags[?
Key=='Environment' && Value=='Production']" --output text)
    if [ ! -z "$TAGS" ]; then
        echo "Table with Production tag: $TABLE_NAME"
    fi
done
```

- For API details, see the following topics in *AWS CLI Command Reference*.
  - CreatePolicy
  - CreateTable
  - ListTables

For a complete list of AWS SDK developer guides and code examples, see Using DynamoDB with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Understand update expression order in DynamoDB with an AWS SDK

The following code examples show how to understand update expression order.

- Learn how DynamoDB processes update expressions.
- Understand the order of operations in update expressions.
- Avoid unexpected results by understanding expression evaluation.

Java

SDK for Java 2.x

Demonstrate update expression order using AWS SDK for Java 2.x.

```
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.GetItemRequest;
import software.amazon.awssdk.services.dynamodb.model.GetItemResponse;
import software.amazon.awssdk.services.dynamodb.model.ReturnValue;
import software.amazon.awssdk.services.dynamodb.model.UpdateItemRequest;
import software.amazon.awssdk.services.dynamodb.model.UpdateItemResponse;

import java.util.HashMap;
import java.util.Map;

    /**
     * Demonstrates the effect of update expression order.
     *
     * <p>This method shows how the order of operations in an update expression
     * affects the result of the update.
     *
     * @param dynamoDbClient The DynamoDB client
     * @param tableName The name of the DynamoDB table
     * @param key The key of the item to update
     * @return Map containing the results of different update orders
     * @throws DynamoDbException if an error occurs during the operation
     */
    public static Map<String, Object> demonstrateUpdateOrder(
        DynamoDbClient dynamoDbClient, String tableName, Map<String,
 AttributeValue> key) {
```

```
        Map<String, Object> results = new HashMap<>();

        try {
            // Initialize the item with a counter
            UpdateItemRequest initRequest = UpdateItemRequest.builder()
                .tableName(tableName)
                .key(key)
                .updateExpression("SET Counter = :zero, OldCounter = :zero")
                .expressionAttributeValues(
                    Map.of(":zero", AttributeValue.builder().n("0").build()))
                .returnValues(ReturnValue.UPDATED_NEW)
                .build();

            dynamoDbClient.updateItem(initRequest);

            // Example 1: SET first, then ADD
            UpdateItemRequest setFirstRequest = UpdateItemRequest.builder()
                .tableName(tableName)
                .key(key)
                .updateExpression("SET Counter = :value ADD
OldCounter :increment")
                .expressionAttributeValues(Map.of(
                    ":value", AttributeValue.builder().n("10").build(),
                    ":increment", AttributeValue.builder().n("5").build()))
                .returnValues(ReturnValue.UPDATED_NEW)
                .build();

            UpdateItemResponse setFirstResponse =
dynamoDbClient.updateItem(setFirstRequest);
            results.put("setFirstResponse", setFirstResponse);

            // Reset the item
            dynamoDbClient.updateItem(initRequest);

            // Example 2: ADD first, then SET
            UpdateItemRequest addFirstRequest = UpdateItemRequest.builder()
                .tableName(tableName)
                .key(key)
                .updateExpression("ADD Counter :increment SET OldCounter
= :value")
                .expressionAttributeValues(Map.of(
                    ":value", AttributeValue.builder().n("10").build(),
                    ":increment", AttributeValue.builder().n("5").build()))
                .returnValues(ReturnValue.UPDATED_NEW)
```

```
                .build();

            UpdateItemResponse addFirstResponse =
dynamoDbClient.updateItem(addFirstRequest);
            results.put("addFirstResponse", addFirstResponse);

            // Reset the item
            dynamoDbClient.updateItem(initRequest);

            // Example 3: SET with multiple attributes
            UpdateItemRequest multiSetRequest = UpdateItemRequest.builder()
                .tableName(tableName)
                .key(key)
                .updateExpression("SET Counter = :value, OldCounter = Counter")
                .expressionAttributeValues(
                    Map.of(":value", AttributeValue.builder().n("10").build()))
                .returnValues(ReturnValue.UPDATED_NEW)
                .build();

            UpdateItemResponse multiSetResponse =
dynamoDbClient.updateItem(multiSetRequest);
            results.put("multiSetResponse", multiSetResponse);

            // Reset the item
            dynamoDbClient.updateItem(initRequest);

            // Example 4: SET with expression using the same attribute
            UpdateItemRequest selfReferenceRequest = UpdateItemRequest.builder()
                .tableName(tableName)
                .key(key)
                .updateExpression("SET Counter = Counter + :increment, OldCounter
= Counter")
                .expressionAttributeValues(
                    Map.of(":increment",
AttributeValue.builder().n("5").build()))
                .returnValues(ReturnValue.UPDATED_NEW)
                .build();

            UpdateItemResponse selfReferenceResponse =
dynamoDbClient.updateItem(selfReferenceRequest);
            results.put("selfReferenceResponse", selfReferenceResponse);

            results.put("success", true);
```

```
        } catch (DynamoDbException e) {
            results.put("success", false);
            results.put("error", e.getMessage());
        }

        return results;
    }

    /**
     * Updates an item with SET first, then REMOVE.
     *
     * <p>This method demonstrates updating an item with SET operation first,
     * followed by a REMOVE operation.
     *
     * @param dynamoDbClient The DynamoDB client
     * @param tableName The name of the DynamoDB table
     * @param key The key of the item to update
     * @param attributeToSet The attribute to set
     * @param setValue The value to set
     * @param attributeToRemove The attribute to remove
     * @return The response from DynamoDB
     * @throws DynamoDbException if an error occurs during the operation
     */
    public static UpdateItemResponse updateWithSetFirst(
        DynamoDbClient dynamoDbClient,
        String tableName,
        Map<String, AttributeValue> key,
        String attributeToSet,
        AttributeValue setValue,
        String attributeToRemove) {

        // Define the update parameters
        UpdateItemRequest request = UpdateItemRequest.builder()
            .tableName(tableName)
            .key(key)
            .updateExpression("SET #setAttr = :setValue REMOVE #removeAttr")
            .expressionAttributeNames(Map.of(
                "#setAttr", attributeToSet,
                "#removeAttr", attributeToRemove))
            .expressionAttributeValues(Map.of(":setValue", setValue))
            .returnValues(ReturnValue.UPDATED_NEW)
            .build();

        // Perform the update operation
```

```
        try {
            return dynamoDbClient.updateItem(request);
        } catch (DynamoDbException e) {
            throw DynamoDbException.builder()
                .message("Failed to update item with SET first: " +
e.getMessage())
                .cause(e)
                .build();
        }
    }

    /**
     * Updates an item with REMOVE first, then SET.
     *
     * <p>This method demonstrates updating an item with REMOVE operation first,
     * followed by a SET operation.
     *
     * @param dynamoDbClient The DynamoDB client
     * @param tableName The name of the DynamoDB table
     * @param key The key of the item to update
     * @param attributeToSet The attribute to set
     * @param setValue The value to set
     * @param attributeToRemove The attribute to remove
     * @return The response from DynamoDB
     * @throws DynamoDbException if an error occurs during the operation
     */
    public static UpdateItemResponse updateWithRemoveFirst(
        DynamoDbClient dynamoDbClient,
        String tableName,
        Map<String, AttributeValue> key,
        String attributeToSet,
        AttributeValue setValue,
        String attributeToRemove) {

        // Define the update parameters
        UpdateItemRequest request = UpdateItemRequest.builder()
            .tableName(tableName)
            .key(key)
            .updateExpression("REMOVE #removeAttr SET #setAttr = :setValue")
            .expressionAttributeNames(Map.of(
                "#setAttr", attributeToSet,
                "#removeAttr", attributeToRemove))
            .expressionAttributeValues(Map.of(":setValue", setValue))
            .returnValues(ReturnValue.UPDATED_NEW)
```

```
            .build();

        // Perform the update operation
        try {
            return dynamoDbClient.updateItem(request);
        } catch (DynamoDbException e) {
            throw DynamoDbException.builder()
                .message("Failed to update item with REMOVE first: " +
e.getMessage())
                .cause(e)
                .build();
        }
    }

    /**
     * Updates an item with all operation types in a specific order.
     *
     * <p>This method demonstrates using all operation types (SET, REMOVE, ADD,
DELETE)
     * in a specific order in a single update expression.
     *
     * @param dynamoDbClient The DynamoDB client
     * @param tableName The name of the DynamoDB table
     * @param key The key of the item to update
     * @return The response from DynamoDB
     * @throws DynamoDbException if an error occurs during the operation
     */
    public static UpdateItemResponse updateWithAllOperationTypes(
        DynamoDbClient dynamoDbClient, String tableName, Map<String,
AttributeValue> key) {

        // Define the update parameters
        UpdateItemRequest request = UpdateItemRequest.builder()
            .tableName(tableName)
            .key(key)
            .updateExpression("SET #stringAttr = :stringVal, #mapAttr.#nestedAttr
= :nestedVal " + "REMOVE #oldAttr "
                + "ADD #counterAttr :increment "
                + "DELETE #stringSetAttr :stringSetVal")
            .expressionAttributeNames(Map.of(
                "#stringAttr", "StringAttribute",
                "#mapAttr", "MapAttribute",
                "#nestedAttr", "NestedAttribute",
                "#oldAttr", "OldAttribute",
```

```
                "#counterAttr", "CounterAttribute",
                "#stringSetAttr", "StringSetAttribute"))
            .expressionAttributeValues(Map.of(
                ":stringVal", AttributeValue.builder().s("New Value").build(),
                ":nestedVal", AttributeValue.builder().s("Nested Value").build(),
                ":increment", AttributeValue.builder().n("1").build(),
                ":stringSetVal", AttributeValue.builder().ss("Value1").build()))
            .returnValues(ReturnValue.UPDATED_NEW)
            .build();

        // Perform the update operation
        try {
            return dynamoDbClient.updateItem(request);
        } catch (DynamoDbException e) {
            throw DynamoDbException.builder()
                .message("Failed to update item with all operation types: " +
e.getMessage())
                .cause(e)
                .build();
        }
    }

    /**
     * Gets the current state of an item.
     *
     * <p>Helper method to retrieve the current state of an item.
     *
     * @param dynamoDbClient The DynamoDB client
     * @param tableName The name of the DynamoDB table
     * @param key The key of the item to get
     * @return The item or null if not found
     * @throws DynamoDbException if an error occurs during the operation
     */
    public static Map<String, AttributeValue> getItem(
        DynamoDbClient dynamoDbClient, String tableName, Map<String,
AttributeValue> key) {

        // Define the get parameters
        GetItemRequest request =
            GetItemRequest.builder().tableName(tableName).key(key).build();

        // Perform the get operation
        try {
            GetItemResponse response = dynamoDbClient.getItem(request);
```

```
            // Return the item if it exists, otherwise null
            return response.item();
        } catch (DynamoDbException e) {
            throw DynamoDbException.builder()
                .message("Failed to get item: " + e.getMessage())
                .cause(e)
                .build();
        }
    }
```

Example usage of update expression order with AWS SDK for Java 2.x.

```
    public static void exampleUsage(DynamoDbClient dynamoDbClient, String
tableName) {
        // Example key
        Map<String, AttributeValue> key = new HashMap<>();
        key.put("ProductId", AttributeValue.builder().s("P12345").build());

        System.out.println("Demonstrating update expression order in DynamoDB");

        try {
            // Example 1: Demonstrate update order effects
            System.out.println("\nExample 1: Demonstrating update order
effects");
            Map<String, Object> orderResults =
demonstrateUpdateOrder(dynamoDbClient, tableName, key);

            if ((boolean) orderResults.get("success")) {
                System.out.println("SET first, then ADD:");
                System.out.println("  " + orderResults.get("setFirstResponse"));

                System.out.println("ADD first, then SET:");
                System.out.println("  " + orderResults.get("addFirstResponse"));

                System.out.println("SET with multiple attributes:");
                System.out.println("  " + orderResults.get("multiSetResponse"));

                System.out.println("SET with self-reference:");
                System.out.println("  " +
orderResults.get("selfReferenceResponse"));
            } else {
```

```
                System.out.println("Error: " + orderResults.get("error"));
            }

            // Example 2: Update with SET first, then REMOVE
            System.out.println("\nExample 2: Update with SET first, then
REMOVE");
            UpdateItemResponse setFirstResponse = updateWithSetFirst(
                dynamoDbClient,
                tableName,
                key,
                "Status",
                AttributeValue.builder().s("Active").build(),
                "OldStatus");

            System.out.println("Updated attributes: " +
setFirstResponse.attributes());

            // Example 3: Update with REMOVE first, then SET
            System.out.println("\nExample 3: Update with REMOVE first, then
SET");
            UpdateItemResponse removeFirstResponse = updateWithRemoveFirst(
                dynamoDbClient,
                tableName,
                key,
                "Status",
                AttributeValue.builder().s("Inactive").build(),
                "OldStatus");

            System.out.println("Updated attributes: " +
removeFirstResponse.attributes());

            // Example 4: Update with all operation types
            System.out.println("\nExample 4: Update with all operation types");
            UpdateItemResponse allOpsResponse =
updateWithAllOperationTypes(dynamoDbClient, tableName, key);

            System.out.println("Updated attributes: " +
allOpsResponse.attributes());

            // Example 5: Get the current state of the item
            System.out.println("\nExample 5: Current state of the item");
            Map<String, AttributeValue> item = getItem(dynamoDbClient, tableName,
key);
```

```
            if (item != null) {
                System.out.println("Item: " + item);
            } else {
                System.out.println("Item not found");
            }

            // Explain update expression order
            System.out.println("\nKey points about update expression order in
 DynamoDB:");
            System.out.println("1. Update expressions are processed in this
 order: SET, REMOVE, ADD, DELETE");
            System.out.println("2. Within each clause, operations are processed
 from left to right");
            System.out.println("3. SET operations use the item state before any
 updates in the expression");
            System.out.println("4. When an attribute is referenced multiple
 times, the first operation wins");
            System.out.println("5. To reference a new value, split the update
 into multiple operations");
            System.out.println("6. The order of clauses in the expression doesn't
 change the evaluation order");
            System.out.println("7. For complex updates, consider using multiple
 separate update operations");

        } catch (DynamoDbException e) {
            System.err.println("Error: " + e.getMessage());
            e.printStackTrace();
        }
    }
```

- For API details, see UpdateItem in *AWS SDK for Java 2.x API Reference*.

JavaScript

SDK for JavaScript (v3)

Demonstrate update expression order using AWS SDK for JavaScript.

```
const { DynamoDBClient } = require("@aws-sdk/client-dynamodb");
const {
  DynamoDBDocumentClient,
  UpdateCommand,
```

```
  GetCommand,
  PutCommand
} = require("@aws-sdk/lib-dynamodb");

/**
 * Update an item with multiple actions in a single update expression.
 *
 * This function demonstrates how to use multiple actions in a single update
 expression
 * and how DynamoDB processes these actions.
 *
 * @param {Object} config - AWS configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {Object} key - The primary key of the item to update
 * @param {string} updateExpression - The update expression with multiple actions
 * @param {Object} [expressionAttributeNames] - Expression attribute name
 placeholders
 * @param {Object} [expressionAttributeValues] - Expression attribute value
 placeholders
 * @returns {Promise<Object>} - The response from DynamoDB
 */
async function updateWithMultipleActions(
  config,
  tableName,
  key,
  updateExpression,
  expressionAttributeNames,
  expressionAttributeValues
) {
  // Initialize the DynamoDB client
  const client = new DynamoDBClient(config);
  const docClient = DynamoDBDocumentClient.from(client);

  // Prepare the update parameters
  const updateParams = {
    TableName: tableName,
    Key: key,
    UpdateExpression: updateExpression,
    ReturnValues: "UPDATED_NEW"
  };

  // Add expression attribute names if provided
  if (expressionAttributeNames) {
    updateParams.ExpressionAttributeNames = expressionAttributeNames;
```

```
  }

  // Add expression attribute values if provided
  if (expressionAttributeValues) {
    updateParams.ExpressionAttributeValues = expressionAttributeValues;
  }

  // Execute the update
  const response = await docClient.send(new UpdateCommand(updateParams));

  return response;
}

/**
 * Demonstrate that variables hold copies of existing values before
 modifications.
 *
 * This function creates an item with initial values, then updates it with an
 expression
 * that uses the values of attributes before they are modified in the same
 expression.
 *
 * @param {Object} config - AWS configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {Object} key - The primary key of the item to create and update
 * @returns {Promise<Object>} - A dictionary containing the results of the
 demonstration
 */
async function demonstrateValueCopying(
  config,
  tableName,
  key
) {
  // Initialize the DynamoDB client
  const client = new DynamoDBClient(config);
  const docClient = DynamoDBDocumentClient.from(client);

  // Step 1: Create an item with initial values
  const initialItem = { ...key, a: 1, b: 2, c: 3 };

  await docClient.send(new PutCommand({
    TableName: tableName,
    Item: initialItem
  }));
```

```
  // Step 2: Get the item to verify initial state
  const responseBefore = await docClient.send(new GetCommand({
    TableName: tableName,
    Key: key
  }));

  const itemBefore = responseBefore.Item || {};

  // Step 3: Update the item with an expression that uses values before they are
 modified
  // This expression removes 'a', then sets 'b' to the value of 'a', and 'c' to
 the value of 'b'
  const updateResponse = await docClient.send(new UpdateCommand({
    TableName: tableName,
    Key: key,
    UpdateExpression: "REMOVE a SET b = a, c = b",
    ReturnValues: "UPDATED_NEW"
  }));

  // Step 4: Get the item to verify final state
  const responseAfter = await docClient.send(new GetCommand({
    TableName: tableName,
    Key: key
  }));

  const itemAfter = responseAfter.Item || {};

  // Return the results
  return {
    initialState: itemBefore,
    updateResponse: updateResponse,
    finalState: itemAfter
  };
}

/**
 * Demonstrate the order in which different action types are processed.
 *
 * This function creates an item with initial values, then updates it with an
 expression
 * that includes multiple action types (SET, REMOVE, ADD, DELETE) to show the
 order
 * in which they are processed.
```

```
 *
 * @param {Object} config - AWS configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {Object} key - The primary key of the item to create and update
 * @returns {Promise<Object>} - A dictionary containing the results of the
 demonstration
 */
async function demonstrateActionOrder(
  config,
  tableName,
  key
) {
  // Initialize the DynamoDB client
  const client = new DynamoDBClient(config);
  const docClient = DynamoDBDocumentClient.from(client);

  // Step 1: Create an item with initial values
  const initialItem = {
    ...key,
    counter: 10,
    set_attr: new Set(["A", "B", "C"]),
    to_remove: "This will be removed",
    to_modify: "Original value"
  };

  await docClient.send(new PutCommand({
    TableName: tableName,
    Item: initialItem
  }));

  // Step 2: Get the item to verify initial state
  const responseBefore = await docClient.send(new GetCommand({
    TableName: tableName,
    Key: key
  }));

  const itemBefore = responseBefore.Item || {};

  // Step 3: Update the item with multiple action types
  // The actions will be processed in this order: REMOVE, SET, ADD, DELETE
  const updateResponse = await docClient.send(new UpdateCommand({
    TableName: tableName,
    Key: key,
```

```
    UpdateExpression: "REMOVE to_remove SET to_modify = :new_value ADD
 counter :increment DELETE set_attr :elements",
    ExpressionAttributeValues: {
      ":new_value": "Updated value",
      ":increment": 5,
      ":elements": new Set(["B"])
    },
    ReturnValues: "UPDATED_NEW"
  }));

  // Step 4: Get the item to verify final state
  const responseAfter = await docClient.send(new GetCommand({
    TableName: tableName,
    Key: key
  }));

  const itemAfter = responseAfter.Item || {};

  // Return the results
  return {
    initialState: itemBefore,
    updateResponse: updateResponse,
    finalState: itemAfter
  };
}

/**
 * Update multiple attributes with a single SET action.
 *
 * This function demonstrates how to update multiple attributes in a single SET
 action,
 * which is more efficient than using multiple separate update operations.
 *
 * @param {Object} config - AWS configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {Object} key - The primary key of the item to update
 * @param {Object} attributes - The attributes to update and their new values
 * @returns {Promise<Object>} - The response from DynamoDB
 */
async function updateWithMultipleSetActions(
  config,
  tableName,
  key,
  attributes
```

```
) {
  // Initialize the DynamoDB client
  const client = new DynamoDBClient(config);
  const docClient = DynamoDBDocumentClient.from(client);

  // Build the update expression and expression attribute values
  let updateExpression = "SET ";
  const expressionAttributeValues = {};

  // Add each attribute to the update expression
  Object.entries(attributes).forEach(([attrName, attrValue], index) => {
    const valuePlaceholder = `:val${index}`;

    if (index > 0) {
      updateExpression += ", ";
    }
    updateExpression += `${attrName} = ${valuePlaceholder}`;

    expressionAttributeValues[valuePlaceholder] = attrValue;
  });

  // Execute the update
  const response = await docClient.send(new UpdateCommand({
    TableName: tableName,
    Key: key,
    UpdateExpression: updateExpression,
    ExpressionAttributeValues: expressionAttributeValues,
    ReturnValues: "UPDATED_NEW"
  }));

  return response;
}

/**
 * Update an attribute with a value from another attribute or a default value.
 *
 * This function demonstrates how to use if_not_exists to conditionally copy a
 value
 * from one attribute to another, or use a default value if the source doesn't
 exist.
 *
 * @param {Object} config - AWS configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {Object} key - The primary key of the item to update
```

```
 * @param {string} sourceAttribute - The attribute to copy the value from
 * @param {string} targetAttribute - The attribute to update
 * @param {any} defaultValue - The default value to use if the source attribute
 doesn't exist
 * @returns {Promise<Object>} - The response from DynamoDB
 */
async function updateWithConditionalValueCopying(
  config,
  tableName,
  key,
  sourceAttribute,
  targetAttribute,
  defaultValue
) {
  // Initialize the DynamoDB client
  const client = new DynamoDBClient(config);
  const docClient = DynamoDBDocumentClient.from(client);

  // Use if_not_exists to conditionally copy the value
  const response = await docClient.send(new UpdateCommand({
    TableName: tableName,
    Key: key,
    UpdateExpression: `SET ${targetAttribute} =
 if_not_exists(${sourceAttribute}, :default)`,
    ExpressionAttributeValues: {
      ":default": defaultValue
    },
    ReturnValues: "UPDATED_NEW"
  }));

  return response;
}

/**
 * Demonstrate complex update expressions with multiple operations on the same
 attribute.
 *
 * This function shows how DynamoDB processes multiple operations on the same
 attribute
 * in a single update expression.
 *
 * @param {Object} config - AWS configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {Object} key - The primary key of the item to create and update
```

```
 * @returns {Promise<Object>} - A dictionary containing the results of the
 demonstration
 */
async function demonstrateMultipleOperationsOnSameAttribute(
  config,
  tableName,
  key
) {
  // Initialize the DynamoDB client
  const client = new DynamoDBClient(config);
  const docClient = DynamoDBDocumentClient.from(client);

  // Step 1: Create an item with initial values
  const initialItem = {
    ...key,
    counter: 10,
    list_attr: [1, 2, 3],
    map_attr: {
      nested1: "value1",
      nested2: "value2"
    }
  };

  await docClient.send(new PutCommand({
    TableName: tableName,
    Item: initialItem
  }));

  // Step 2: Get the item to verify initial state
  const responseBefore = await docClient.send(new GetCommand({
    TableName: tableName,
    Key: key
  }));

  const itemBefore = responseBefore.Item || {};

  // Step 3: Update the item with multiple operations on the same attributes
  const updateResponse = await docClient.send(new UpdateCommand({
    TableName: tableName,
    Key: key,
    UpdateExpression: `
      SET counter = counter + :inc1,
          counter = counter + :inc2,
          map_attr.nested1 = :new_val1,
```

```
          map_attr.nested3 = :new_val3,
          list_attr[0] = list_attr[1],
          list_attr[1] = list_attr[2]
    `,
    ExpressionAttributeValues: {
      ":inc1": 5,
      ":inc2": 3,
      ":new_val1": "updated_value1",
      ":new_val3": "new_value3"
    },
    ReturnValues: "UPDATED_NEW"
  }));

  // Step 4: Get the item to verify final state
  const responseAfter = await docClient.send(new GetCommand({
    TableName: tableName,
    Key: key
  }));

  const itemAfter = responseAfter.Item || {};

  // Return the results
  return {
    initialState: itemBefore,
    updateResponse: updateResponse,
    finalState: itemAfter
  };
}
```

Example usage of update expression order with AWS SDK for JavaScript.

```
/**
 * Example of how to use update expression order of operations in DynamoDB.
 */
async function exampleUsage() {
  // Example parameters
  const config = { region: "us-west-2" };
  const tableName = "OrderProcessing";

  console.log("Demonstrating update expression order of operations in DynamoDB");

  try {
```

```
   // Example 1: Demonstrating value copying in update expressions
   console.log("\nExample 1: Demonstrating value copying in update
expressions");
   const results1 = await demonstrateValueCopying(
     config,
     tableName,
     { OrderId: "order123" }
   );

   console.log("Initial state:", JSON.stringify(results1.initialState, null,
2));
   console.log("Update response:", JSON.stringify(results1.updateResponse, null,
2));
   console.log("Final state:", JSON.stringify(results1.finalState, null, 2));

   console.log("\nExplanation:");
   console.log("1. The initial state had a=1, b=2, c=3");
   console.log("2. The update expression 'REMOVE a SET b = a, c = b' did the
following:");
   console.log("   - Copied the value of 'a' (which was 1) to be used for 'b'");
   console.log("   - Copied the value of 'b' (which was 2) to be used for 'c'");
   console.log("   - Removed the attribute 'a'");
   console.log("3. The final state has b=1, c=2, and 'a' is removed");
   console.log("4. This demonstrates that DynamoDB uses the values of attributes
as they were BEFORE any modifications");

   // Example 2: Demonstrating the order of different action types
   console.log("\nExample 2: Demonstrating the order of different action
types");
   const results2 = await demonstrateActionOrder(
     config,
     tableName,
     { OrderId: "order456" }
   );

   console.log("Initial state:", JSON.stringify(results2.initialState, null,
2));
   console.log("Update response:", JSON.stringify(results2.updateResponse, null,
2));
   console.log("Final state:", JSON.stringify(results2.finalState, null, 2));

   console.log("\nExplanation:");
   console.log("1. The update expression contained multiple action types:
REMOVE, SET, ADD, DELETE");
```

```
   console.log("2. DynamoDB processes these actions in this order: REMOVE, SET,
ADD, DELETE");
   console.log("3. First, 'to_remove' was removed");
   console.log("4. Then, 'to_modify' was set to a new value");
   console.log("5. Next, 'counter' was incremented by 5");
   console.log("6. Finally, 'B' was removed from the set attribute");

   // Example 3: Updating multiple attributes in a single SET action
   console.log("\nExample 3: Updating multiple attributes in a single SET
action");
   const response3 = await updateWithMultipleSetActions(
     config,
     tableName,
     { OrderId: "order789" },
     {
       Status: "Shipped",
       ShippingDate: "2025-05-28",
       TrackingNumber: "1Z999AA10123456784"
     }
   );

   console.log("Multiple attributes updated successfully:",
JSON.stringify(response3.Attributes, null, 2));

   // Example 4: Conditional value copying with if_not_exists
   console.log("\nExample 4: Conditional value copying with if_not_exists");
   const response4 = await updateWithConditionalValueCopying(
     config,
     tableName,
     { OrderId: "order101" },
     "PreferredShippingMethod",
     "ShippingMethod",
     "Standard"
   );

   console.log("Conditional value copying result:",
JSON.stringify(response4.Attributes, null, 2));

   // Example 5: Multiple operations on the same attribute
   console.log("\nExample 5: Multiple operations on the same attribute");
   const results5 = await demonstrateMultipleOperationsOnSameAttribute(
     config,
     tableName,
     { OrderId: "order202" }
```

```
    );

    console.log("Initial state:", JSON.stringify(results5.initialState, null,
 2));
    console.log("Update response:", JSON.stringify(results5.updateResponse, null,
 2));
    console.log("Final state:", JSON.stringify(results5.finalState, null, 2));

    console.log("\nExplanation:");
    console.log("1. The counter was incremented twice (first by 5, then by 3) for
 a total of +8");
    console.log("2. The map attribute had one value updated and a new nested
 attribute added");
    console.log("3. The list attribute had values shifted (value at index 1 moved
 to index 0, value at index 2 moved to index 1)");
    console.log("4. All operations within the SET action are processed from left
 to right");

    // Key points about update expression order of operations
    console.log("\nKey Points About Update Expression Order of Operations:");
    console.log("1. Variables in expressions hold copies of attribute values as
 they existed BEFORE any modifications");
    console.log("2. Multiple actions in an update expression are processed in
 this order: REMOVE, SET, ADD, DELETE");
    console.log("3. Within each action type, operations are processed from left
 to right");
    console.log("4. You can reference the same attribute multiple times in an
 expression");
    console.log("5. You can use if_not_exists() to conditionally set values based
 on attribute existence");
    console.log("6. Using a single update expression with multiple actions is
 more efficient than multiple separate updates");
    console.log("7. The update expression is atomic - either all actions succeed
 or none do");

  } catch (error) {
    console.error("Error:", error);
  }
}
```

- For API details, see UpdateItem in *AWS SDK for JavaScript API Reference*.

Python

SDK for Python (Boto3)

Demonstrate update expression order using AWS SDK for Python (Boto3).

```
import boto3
import json
from typing import Any, Dict, Optional


def update_with_multiple_actions(
    table_name: str,
    key: Dict[str, Any],
    update_expression: str,
    expression_attribute_names: Optional[Dict[str, str]] = None,
    expression_attribute_values: Optional[Dict[str, Any]] = None,
) -> Dict[str, Any]:
    """
    Update an item with multiple actions in a single update expression.

    This function demonstrates how to use multiple actions in a single update
 expression
    and how DynamoDB processes these actions.

    Args:
        table_name (str): The name of the DynamoDB table.
        key (Dict[str, Any]): The primary key of the item to update.
        update_expression (str): The update expression with multiple actions.
        expression_attribute_names (Optional[Dict[str, str]]): Expression
 attribute name placeholders.
        expression_attribute_values (Optional[Dict[str, Any]]): Expression
 attribute value placeholders.

    Returns:
        Dict[str, Any]: The response from DynamoDB containing the updated
 attribute values.
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # Prepare the update parameters
    update_params = {
```

```
        "Key": key,
        "UpdateExpression": update_expression,
        "ReturnValues": "UPDATED_NEW",
    }

    # Add expression attribute names if provided
    if expression_attribute_names:
        update_params["ExpressionAttributeNames"] = expression_attribute_names

    # Add expression attribute values if provided
    if expression_attribute_values:
        update_params["ExpressionAttributeValues"] = expression_attribute_values

    # Execute the update
    response = table.update_item(**update_params)

    return response


def demonstrate_value_copying(table_name: str, key: Dict[str, Any]) -> Dict[str,
 Any]:
    """
    Demonstrate that variables hold copies of existing values before
 modifications.

    This function creates an item with initial values, then updates it with an
 expression
    that uses the values of attributes before they are modified in the same
 expression.

    Args:
        table_name (str): The name of the DynamoDB table.
        key (Dict[str, Any]): The primary key of the item to create and update.

    Returns:
        Dict[str, Any]: A dictionary containing the results of the demonstration.
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # Step 1: Create an item with initial values
    initial_item = key.copy()
    initial_item.update({"a": 1, "b": 2, "c": 3})
```

```
    table.put_item(Item=initial_item)

    # Step 2: Get the item to verify initial state
    response_before = table.get_item(Key=key)
    item_before = response_before.get("Item", {})

    # Step 3: Update the item with an expression that uses values before they are
 modified
    # This expression removes 'a', then sets 'b' to the value of 'a', and 'c' to
 the value of 'b'
    update_response = table.update_item(
        Key=key, UpdateExpression="REMOVE a SET b = a, c = b",
 ReturnValues="UPDATED_NEW"
    )

    # Step 4: Get the item to verify final state
    response_after = table.get_item(Key=key)
    item_after = response_after.get("Item", {})

    # Return the results
    return {
        "initial_state": item_before,
        "update_response": update_response,
        "final_state": item_after,
    }


def demonstrate_action_order(table_name: str, key: Dict[str, Any]) -> Dict[str,
 Any]:
    """
    Demonstrate the order in which different action types are processed.

    This function creates an item with initial values, then updates it with an
 expression
    that includes multiple action types (SET, REMOVE, ADD, DELETE) to show the
 order
    in which they are processed.

    Args:
        table_name (str): The name of the DynamoDB table.
        key (Dict[str, Any]): The primary key of the item to create and update.

    Returns:
```

```
        Dict[str, Any]: A dictionary containing the results of the demonstration.
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # Step 1: Create an item with initial values
    initial_item = key.copy()
    initial_item.update(
        {
            "counter": 10,
            "set_attr": set(["A", "B", "C"]),
            "to_remove": "This will be removed",
            "to_modify": "Original value",
        }
    )

    table.put_item(Item=initial_item)

    # Step 2: Get the item to verify initial state
    response_before = table.get_item(Key=key)
    item_before = response_before.get("Item", {})

    # Step 3: Update the item with multiple action types
    # The actions will be processed in this order: REMOVE, SET, ADD, DELETE
    update_response = table.update_item(
        Key=key,
        UpdateExpression="REMOVE to_remove SET to_modify = :new_value ADD
counter :increment DELETE set_attr :elements",
        ExpressionAttributeValues={
            ":new_value": "Updated value",
            ":increment": 5,
            ":elements": set(["B"]),
        },
        ReturnValues="UPDATED_NEW",
    )

    # Step 4: Get the item to verify final state
    response_after = table.get_item(Key=key)
    item_after = response_after.get("Item", {})

    # Return the results
    return {
        "initial_state": item_before,
```

```
        "update_response": update_response,
        "final_state": item_after,
    }


def update_with_multiple_set_actions(
    table_name: str, key: Dict[str, Any], attributes: Dict[str, Any]
) -> Dict[str, Any]:
    """
    Update multiple attributes with a single SET action.

    This function demonstrates how to update multiple attributes in a single SET
 action,
    which is more efficient than using multiple separate update operations.

    Args:
        table_name (str): The name of the DynamoDB table.
        key (Dict[str, Any]): The primary key of the item to update.
        attributes (Dict[str, Any]): The attributes to update and their new
 values.

    Returns:
        Dict[str, Any]: The response from DynamoDB containing the updated
 attribute values.
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # Build the update expression and expression attribute values
    update_expression = "SET "
    expression_attribute_values = {}

    # Add each attribute to the update expression
    for i, (attr_name, attr_value) in enumerate(attributes.items()):
        value_placeholder = f":val{i}"

        if i > 0:
            update_expression += ", "
        update_expression += f"{attr_name} = {value_placeholder}"

        expression_attribute_values[value_placeholder] = attr_value

    # Execute the update
```

```
    response = table.update_item(
        Key=key,
        UpdateExpression=update_expression,
        ExpressionAttributeValues=expression_attribute_values,
        ReturnValues="UPDATED_NEW",
    )

    return response


def update_with_conditional_value_copying(
    table_name: str,
    key: Dict[str, Any],
    source_attribute: str,
    target_attribute: str,
    default_value: Any,
) -> Dict[str, Any]:
    """
    Update an attribute with a value from another attribute or a default value.

    This function demonstrates how to use if_not_exists to conditionally copy a
 value
    from one attribute to another, or use a default value if the source doesn't
 exist.

    Args:
        table_name (str): The name of the DynamoDB table.
        key (Dict[str, Any]): The primary key of the item to update.
        source_attribute (str): The attribute to copy the value from.
        target_attribute (str): The attribute to update.
        default_value (Any): The default value to use if the source attribute
 doesn't exist.

    Returns:
        Dict[str, Any]: The response from DynamoDB containing the updated
 attribute values.
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # Use if_not_exists to conditionally copy the value
    response = table.update_item(
        Key=key,
```

```
        UpdateExpression=f"SET {target_attribute} =
 if_not_exists({source_attribute}, :default)",
        ExpressionAttributeValues={":default": default_value},
        ReturnValues="UPDATED_NEW",
    )

    return response
```

Example usage of update expression order with AWS SDK for Python (Boto3).

```
def example_usage():
    """Example of how to use update expression order of operations in
 DynamoDB."""
    # Example parameters
    table_name = "OrderProcessing"
    key = {"OrderId": "order123"}

    print("Example 1: Demonstrating value copying in update expressions")
    try:
        results = demonstrate_value_copying(table_name=table_name, key=key)

        print(f"Initial state: {json.dumps(results['initial_state'],
 default=str)}")
        print(f"Update response: {json.dumps(results['update_response'],
 default=str)}")
        print(f"Final state: {json.dumps(results['final_state'], default=str)}")

        print("\nExplanation:")
        print("1. The initial state had a=1, b=2, c=3")
        print("2. The update expression 'REMOVE a SET b = a, c = b' did the
 following:")
        print("   - Copied the value of 'a' (which was 1) to be used for 'b'")
        print("   - Copied the value of 'b' (which was 2) to be used for 'c'")
        print("   - Removed the attribute 'a'")
        print("3. The final state has b=1, c=2, and 'a' is removed")
        print(
            "4. This demonstrates that DynamoDB uses the values of attributes as
 they were BEFORE any modifications"
        )
    except Exception as e:
```

```
        print(f"Error demonstrating value copying: {e}")

    print("\nExample 2: Demonstrating the order of different action types")
    try:
        results = demonstrate_action_order(table_name=table_name, key={"OrderId":
"order456"})

        print(f"Initial state: {json.dumps(results['initial_state'],
default=str)}")
        print(f"Update response: {json.dumps(results['update_response'],
default=str)}")
        print(f"Final state: {json.dumps(results['final_state'], default=str)}")

        print("\nExplanation:")
        print("1. The update expression contained multiple action types: REMOVE,
SET, ADD, DELETE")
        print("2. DynamoDB processes these actions in this order: REMOVE, SET,
ADD, DELETE")
        print("3. First, 'to_remove' was removed")
        print("4. Then, 'to_modify' was set to a new value")
        print("5. Next, 'counter' was incremented by 5")
        print("6. Finally, 'B' was removed from the set attribute")
    except Exception as e:
        print(f"Error demonstrating action order: {e}")

    print("\nExample 3: Updating multiple attributes in a single SET action")
    try:
        response = update_with_multiple_set_actions(
            table_name=table_name,
            key={"OrderId": "order789"},
            attributes={
                "Status": "Shipped",
                "ShippingDate": "2025-05-14",
                "TrackingNumber": "1Z999AA10123456784",
            },
        )

        print(
            f"Multiple attributes updated successfully:
{json.dumps(response.get('Attributes', {}), default=str)}"
        )
    except Exception as e:
        print(f"Error updating multiple attributes: {e}")
```

```
    print("\nExample 4: Conditional value copying with if_not_exists")
    try:
        response = update_with_conditional_value_copying(
            table_name=table_name,
            key={"OrderId": "order101"},
            source_attribute="PreferredShippingMethod",
            target_attribute="ShippingMethod",
            default_value="Standard",
        )

        print(
            f"Conditional value copying result:
{json.dumps(response.get('Attributes', {}), default=str)}"
        )
    except Exception as e:
        print(f"Error with conditional value copying: {e}")

    print("\nKey Points About Update Expression Order of Operations:")
    print(
        "1. Variables in expressions hold copies of attribute values as they
existed BEFORE any modifications"
    )
    print(
        "2. Multiple actions in an update expression are processed in this order:
REMOVE, SET, ADD, DELETE"
    )
    print("3. Within each action type, operations are processed from left to
right")
    print("4. You can reference the same attribute multiple times in an
expression")
    print("5. You can use if_not_exists() to conditionally set values based on
attribute existence")
    print(
        "6. Using a single update expression with multiple actions is more
efficient than multiple separate updates"
    )
    print("7. The update expression is atomic - either all actions succeed or
none do")
```

- For API details, see UpdateItem in *AWS SDK for Python (Boto3) API Reference*.

For a complete list of AWS SDK developer guides and code examples, see Using DynamoDB with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Update a DynamoDB table setting with warm throughput using an AWS SDK

The following code examples show how to update a table's warm throughput setting.

Java

SDK for Java 2.x

Update warm throughput setting on an existing DynamoDB table using AWS SDK for Java 2.x.

```
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.GlobalSecondaryIndexUpdate;
import
 software.amazon.awssdk.services.dynamodb.model.UpdateGlobalSecondaryIndexAction;
import software.amazon.awssdk.services.dynamodb.model.UpdateTableRequest;
import software.amazon.awssdk.services.dynamodb.model.WarmThroughput;

    public static WarmThroughput buildWarmThroughput(final Long
 readUnitsPerSecond, final Long writeUnitsPerSecond) {
        return WarmThroughput.builder()
            .readUnitsPerSecond(readUnitsPerSecond)
            .writeUnitsPerSecond(writeUnitsPerSecond)
            .build();
    }

    /**
     * Updates a DynamoDB table with warm throughput settings for both the table
 and a global secondary index.
     *
     * @param ddb The DynamoDB client
     * @param tableName The name of the table to update
     * @param tableReadUnitsPerSecond Read units per second for the table
     * @param tableWriteUnitsPerSecond Write units per second for the table
     * @param globalSecondaryIndexName The name of the global secondary index to
 update
```

```
    * @param globalSecondaryIndexReadUnitsPerSecond Read units per second for
the GSI
    * @param globalSecondaryIndexWriteUnitsPerSecond Write units per second for
the GSI
    */
   public static void updateDynamoDBTable(
       final DynamoDbClient ddb,
       final String tableName,
       final Long tableReadUnitsPerSecond,
       final Long tableWriteUnitsPerSecond,
       final String globalSecondaryIndexName,
       final Long globalSecondaryIndexReadUnitsPerSecond,
       final Long globalSecondaryIndexWriteUnitsPerSecond) {

       final WarmThroughput tableWarmThroughput =
           buildWarmThroughput(tableReadUnitsPerSecond,
tableWriteUnitsPerSecond);
       final WarmThroughput gsiWarmThroughput =
           buildWarmThroughput(globalSecondaryIndexReadUnitsPerSecond,
globalSecondaryIndexWriteUnitsPerSecond);

       final GlobalSecondaryIndexUpdate globalSecondaryIndexUpdate =
GlobalSecondaryIndexUpdate.builder()
           .update(UpdateGlobalSecondaryIndexAction.builder()
               .indexName(globalSecondaryIndexName)
               .warmThroughput(gsiWarmThroughput)
               .build())
           .build();

       final UpdateTableRequest request = UpdateTableRequest.builder()
           .tableName(tableName)
           .globalSecondaryIndexUpdates(globalSecondaryIndexUpdate)
           .warmThroughput(tableWarmThroughput)
           .build();

       try {
           ddb.updateTable(request);
       } catch (DynamoDbException e) {
           System.err.println(e.getMessage());
           throw e;
       }

       System.out.println(SUCCESS_MESSAGE);
   }
```

- For API details, see UpdateTable in *AWS SDK for Java 2.x API Reference*.

JavaScript

SDK for JavaScript (v3)

Update warm throughput setting on an existing DynamoDB table using AWS SDK for JavaScript.

```
import { DynamoDBClient, UpdateTableCommand } from "@aws-sdk/client-dynamodb";

export async function updateDynamoDBTableWarmThroughput(
  tableName,
  tableReadUnits,
  tableWriteUnits,
  gsiName,
  gsiReadUnits,
  gsiWriteUnits,
  region = "us-east-1"
) {
  try {
    const ddbClient = new DynamoDBClient({ region: region });

    // Construct the update table request
    const updateTableRequest = {
      TableName: tableName,
      GlobalSecondaryIndexUpdates: [
        {
            Update: {
                IndexName: gsiName,
                WarmThroughput: {
                    ReadUnitsPerSecond: gsiReadUnits,
                    WriteUnitsPerSecond: gsiWriteUnits,
                },
            },
        },
      ],
      WarmThroughput: {
          ReadUnitsPerSecond: tableReadUnits,
          WriteUnitsPerSecond: tableWriteUnits,
      },
```

```
    };

    const command = new UpdateTableCommand(updateTableRequest);
    const response = await ddbClient.send(command);
    console.log(`Table updated successfully! Response:
 ${JSON.stringify(response)}`);
    return response;
  } catch (error) {
    console.error(`Error updating table: ${error}`);
    throw error;
  }
}

// Example usage (commented out for testing)
/*
updateDynamoDBTableWarmThroughput(
  'example-table',
  5, 5,
  'example-index',
  2, 2
);
*/
```

- For API details, see UpdateTable in *AWS SDK for JavaScript API Reference*.

Python

SDK for Python (Boto3)

Update warm throughput setting on an existing DynamoDB table using AWS SDK for Python (Boto3).

```
from boto3 import client
from botocore.exceptions import ClientError


def update_dynamodb_table_warm_throughput(
    table_name,
    table_read_units,
    table_write_units,
    gsi_name,
```

```
    gsi_read_units,
    gsi_write_units,
    region_name="us-east-1",
):
    """
    Updates the warm throughput of a DynamoDB table and a global secondary index.

    :param table_name: The name of the table to update.
    :param table_read_units: The new read units per second for the table's warm
 throughput.
    :param table_write_units: The new write units per second for the table's warm
 throughput.
    :param gsi_name: The name of the global secondary index to update.
    :param gsi_read_units: The new read units per second for the GSI's warm
 throughput.
    :param gsi_write_units: The new write units per second for the GSI's warm
 throughput.
    :param region_name: The AWS Region name to target. defaults to us-east-1
    :return: The response from the update_table operation
    """
    try:
        ddb = client("dynamodb", region_name=region_name)

        # Update the table's warm throughput
        table_warm_throughput = {
            "ReadUnitsPerSecond": table_read_units,
            "WriteUnitsPerSecond": table_write_units,
        }

        # Update the global secondary index's warm throughput
        gsi_warm_throughput = {
            "ReadUnitsPerSecond": gsi_read_units,
            "WriteUnitsPerSecond": gsi_write_units,
        }

        # Construct the global secondary index update
        global_secondary_index_update = [
            {"Update": {"IndexName": gsi_name, "WarmThroughput":
 gsi_warm_throughput}}
        ]

        # Construct the update table request
        update_table_request = {
            "TableName": table_name,
```

```
            "GlobalSecondaryIndexUpdates": global_secondary_index_update,
            "WarmThroughput": table_warm_throughput,
        }

        # Update the table
        response = ddb.update_table(**update_table_request)
        print("Table updated successfully!")
        return response  # Make sure to return the response
    except ClientError as e:
        print(f"Error updating table: {e}")
        raise e
```

- For API details, see UpdateTable in *AWS SDK for Python (Boto3) API Reference*.

For a complete list of AWS SDK developer guides and code examples, see Using DynamoDB with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Update a DynamoDB item with a TTL using an AWS SDK

The following code examples show how to update an item's TTL.

Java

SDK for Java 2.x

Update TTL on an existing DynamoDB item in a table.

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.ResourceNotFoundException;
import software.amazon.awssdk.services.dynamodb.model.UpdateItemRequest;
import software.amazon.awssdk.services.dynamodb.model.UpdateItemResponse;

import java.util.HashMap;
import java.util.Map;
import java.util.Optional;

    public UpdateItemResponse updateItemWithTTL(
```

```
        final String tableName, final String primaryKeyValue, final String
sortKeyValue) {
        // Get current time in epoch second format
        final long currentTime = System.currentTimeMillis() / 1000;

        // Calculate expiration time 90 days from now in epoch second format
        final long expireDate = currentTime + (DAYS_TO_EXPIRE * SECONDS_PER_DAY);

        // Create the key map for the item to update
        final Map<String, AttributeValue> keyMap = new HashMap<>();
        keyMap.put(PRIMARY_KEY_ATTR,
AttributeValue.builder().s(primaryKeyValue).build());
        keyMap.put(SORT_KEY_ATTR,
AttributeValue.builder().s(sortKeyValue).build());

        // Create the expression attribute values
        final Map<String, AttributeValue> expressionAttributeValues = new
HashMap<>();
        expressionAttributeValues.put(
            ":c",
AttributeValue.builder().n(String.valueOf(currentTime)).build());
        expressionAttributeValues.put(
            ":e",
AttributeValue.builder().n(String.valueOf(expireDate)).build());

        final UpdateItemRequest request = UpdateItemRequest.builder()
            .tableName(tableName)
            .key(keyMap)
            .updateExpression(UPDATE_EXPRESSION)
            .expressionAttributeValues(expressionAttributeValues)
            .build();

        try {
            final UpdateItemResponse response =
dynamoDbClient.updateItem(request);
            System.out.println(String.format(SUCCESS_MESSAGE, tableName));
            return response;
        } catch (ResourceNotFoundException e) {
            System.err.format(TABLE_NOT_FOUND_ERROR, tableName);
            throw e;
        } catch (DynamoDbException e) {
            System.err.println(e.getMessage());
            throw e;
        }
```

```
    }
```

- For API details, see UpdateItem in *AWS SDK for Java 2.x API Reference*.

JavaScript

SDK for JavaScript (v3)

```
import { DynamoDBClient, UpdateItemCommand } from "@aws-sdk/client-dynamodb";
import { marshall, unmarshall } from "@aws-sdk/util-dynamodb";

export const updateItem = async (tableName, partitionKey, sortKey, region = 'us-
east-1') => {
    const client = new DynamoDBClient({
        region: region,
        endpoint: `https://dynamodb.${region}.amazonaws.com`
    });

    const currentTime = Math.floor(Date.now() / 1000);
    const expireAt = Math.floor((Date.now() + 90 * 24 * 60 * 60 * 1000) / 1000);

    const params = {
        TableName: tableName,
        Key: marshall({
            partitionKey: partitionKey,
            sortKey: sortKey
        }),
        UpdateExpression: "SET updatedAt = :c, expireAt = :e",
        ExpressionAttributeValues: marshall({
            ":c": currentTime,
            ":e": expireAt
        }),
    };

    try {
        const data = await client.send(new UpdateItemCommand(params));
        const responseData = unmarshall(data.Attributes);
        console.log("Item updated successfully: %s", responseData);
        return responseData;
    } catch (err) {
        console.error("Error updating item:", err);
        throw err;
```

```
    }
}

// Example usage (commented out for testing)
// updateItem('your-table-name', 'your-partition-key-value', 'your-sort-key-
value');
```

- For API details, see UpdateItem in *AWS SDK for JavaScript API Reference*.

Python

SDK for Python (Boto3)

```
from datetime import datetime, timedelta

import boto3


def update_dynamodb_item(table_name, region, primary_key, sort_key):
    """
    Update an existing DynamoDB item with a TTL.
    :param table_name: Name of the DynamoDB table
    :param region: AWS Region of the table - example `us-east-1`
    :param primary_key: one attribute known as the partition key.
    :param sort_key: Also known as a range attribute.
    :return: Void (nothing)
    """
    try:
        # Create the DynamoDB resource.
        dynamodb = boto3.resource("dynamodb", region_name=region)
        table = dynamodb.Table(table_name)

        # Get the current time in epoch second format
        current_time = int(datetime.now().timestamp())

        # Calculate the expireAt time (90 days from now) in epoch second format
        expire_at = int((datetime.now() + timedelta(days=90)).timestamp())

        table.update_item(
            Key={"partitionKey": primary_key, "sortKey": sort_key},
            UpdateExpression="set updatedAt=:c, expireAt=:e",
```

```
            ExpressionAttributeValues={":c": current_time, ":e": expire_at},
        )

        print("Item updated successfully.")
    except Exception as e:
        print(f"Error updating item: {e}")


# Replace with your own values
update_dynamodb_item(
    "your-table-name", "us-west-2", "your-partition-key-value", "your-sort-key-
value"
)
```

- For API details, see UpdateItem in *AWS SDK for Python (Boto3) API Reference*.

For a complete list of AWS SDK developer guides and code examples, see Using DynamoDB with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Update DynamoDB data using PartiQL UPDATE statements with an AWS SDK

The following code example shows how to update data using PartiQL UPDATE statements.

JavaScript

SDK for JavaScript (v3)

Update items in a DynamoDB table using PartiQL UPDATE statements with AWS SDK for JavaScript.

```
/**
 * This example demonstrates how to update items in a DynamoDB table using
 PartiQL.
 * It shows different ways to update documents with various index types.
 */
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
import {
  DynamoDBDocumentClient,
```

```
  ExecuteStatementCommand,
  BatchExecuteStatementCommand,
} from "@aws-sdk/lib-dynamodb";

/**
 * Update a single attribute of an item using PartiQL.
 *
 * @param tableName - The name of the DynamoDB table
 * @param partitionKeyName - The name of the partition key attribute
 * @param partitionKeyValue - The value of the partition key
 * @param attributeName - The name of the attribute to update
 * @param attributeValue - The new value for the attribute
 * @returns The response from the ExecuteStatementCommand
 */
export const updateSingleAttribute = async (
  tableName: string,
  partitionKeyName: string,
  partitionKeyValue: string | number,
  attributeName: string,
  attributeValue: any
) => {
  const client = new DynamoDBClient({});
  const docClient = DynamoDBDocumentClient.from(client);

  const params = {
    Statement: `UPDATE "${tableName}" SET ${attributeName} = ? WHERE
 ${partitionKeyName} = ?`,
    Parameters: [attributeValue, partitionKeyValue],
  };

  try {
    const data = await docClient.send(new ExecuteStatementCommand(params));
    console.log("Item updated successfully");
    return data;
  } catch (err) {
    console.error("Error updating item:", err);
    throw err;
  }
};

/**
 * Update multiple attributes of an item using PartiQL.
 *
 * @param tableName - The name of the DynamoDB table
```

```
 * @param partitionKeyName - The name of the partition key attribute
 * @param partitionKeyValue - The value of the partition key
 * @param attributeUpdates - Object containing attribute names and their new
 values
 * @returns The response from the ExecuteStatementCommand
 */
export const updateMultipleAttributes = async (
  tableName: string,
  partitionKeyName: string,
  partitionKeyValue: string | number,
  attributeUpdates: Record<string, any>
) => {
  const client = new DynamoDBClient({});
  const docClient = DynamoDBDocumentClient.from(client);

  // Create SET clause for each attribute
  const setClause = Object.keys(attributeUpdates)
    .map((attr, index) => `${attr} = ?`)
    .join(", ");

  // Create parameters array with attribute values followed by the partition key
 value
  const parameters = [...Object.values(attributeUpdates), partitionKeyValue];

  const params = {
    Statement: `UPDATE "${tableName}" SET ${setClause} WHERE ${partitionKeyName}
 = ?`,
    Parameters: parameters,
  };

  try {
    const data = await docClient.send(new ExecuteStatementCommand(params));
    console.log("Item updated successfully");
    return data;
  } catch (err) {
    console.error("Error updating item:", err);
    throw err;
  }
};

/**
 * Update an item identified by a composite key (partition key + sort key) using
 PartiQL.
 *
```

```
 * @param tableName - The name of the DynamoDB table
 * @param partitionKeyName - The name of the partition key attribute
 * @param partitionKeyValue - The value of the partition key
 * @param sortKeyName - The name of the sort key attribute
 * @param sortKeyValue - The value of the sort key
 * @param attributeName - The name of the attribute to update
 * @param attributeValue - The new value for the attribute
 * @returns The response from the ExecuteStatementCommand
 */
export const updateItemWithCompositeKey = async (
  tableName: string,
  partitionKeyName: string,
  partitionKeyValue: string | number,
  sortKeyName: string,
  sortKeyValue: string | number,
  attributeName: string,
  attributeValue: any
) => {
  const client = new DynamoDBClient({});
  const docClient = DynamoDBDocumentClient.from(client);

  const params = {
    Statement: `UPDATE "${tableName}" SET ${attributeName} = ? WHERE
 ${partitionKeyName} = ? AND ${sortKeyName} = ?`,
    Parameters: [attributeValue, partitionKeyValue, sortKeyValue],
  };

  try {
    const data = await docClient.send(new ExecuteStatementCommand(params));
    console.log("Item updated successfully");
    return data;
  } catch (err) {
    console.error("Error updating item:", err);
    throw err;
  }
};

/**
 * Update an item with a condition to ensure the update only happens if a
 condition is met.
 *
 * @param tableName - The name of the DynamoDB table
 * @param partitionKeyName - The name of the partition key attribute
 * @param partitionKeyValue - The value of the partition key
```

```
 * @param attributeName - The name of the attribute to update
 * @param attributeValue - The new value for the attribute
 * @param conditionAttribute - The attribute to check in the condition
 * @param conditionValue - The value to compare against in the condition
 * @returns The response from the ExecuteStatementCommand
 */
export const updateItemWithCondition = async (
  tableName: string,
  partitionKeyName: string,
  partitionKeyValue: string | number,
  attributeName: string,
  attributeValue: any,
  conditionAttribute: string,
  conditionValue: any
) => {
  const client = new DynamoDBClient({});
  const docClient = DynamoDBDocumentClient.from(client);

  const params = {
    Statement: `UPDATE "${tableName}" SET ${attributeName} = ? WHERE
 ${partitionKeyName} = ? AND ${conditionAttribute} = ?`,
    Parameters: [attributeValue, partitionKeyValue, conditionValue],
  };

  try {
    const data = await docClient.send(new ExecuteStatementCommand(params));
    console.log("Item updated with condition successfully");
    return data;
  } catch (err) {
    console.error("Error updating item with condition:", err);
    throw err;
  }
};

/**
 * Batch update multiple items using PartiQL.
 *
 * @param tableName - The name of the DynamoDB table
 * @param updates - Array of objects containing key and update information
 * @returns The response from the BatchExecuteStatementCommand
 */
export const batchUpdateItems = async (
  tableName: string,
  updates: Array<{
```

```
    partitionKeyName: string;
    partitionKeyValue: string | number;
    attributeName: string;
    attributeValue: any;
  }>
) => {
  const client = new DynamoDBClient({});
  const docClient = DynamoDBDocumentClient.from(client);

  // Create statements for each update
  const statements = updates.map((update) => {
    return {
      Statement: `UPDATE "${tableName}" SET ${update.attributeName} = ? WHERE
 ${update.partitionKeyName} = ?`,
      Parameters: [update.attributeValue, update.partitionKeyValue],
    };
  });

  const params = {
    Statements: statements,
  };

  try {
    const data = await docClient.send(new BatchExecuteStatementCommand(params));
    console.log("Items batch updated successfully");
    return data;
  } catch (err) {
    console.error("Error batch updating items:", err);
    throw err;
  }
};

/**
 * Example usage showing how to update items with different index types
 */
export const updateExamples = async () => {
  // Update a single attribute using a simple primary key
  await updateSingleAttribute("UsersTable", "userId", "user123", "email",
 "newemail@example.com");

  // Update multiple attributes at once
  await updateMultipleAttributes("UsersTable", "userId", "user123", {
    email: "newemail@example.com",
    name: "John Smith",
```

```
    lastLogin: new Date().toISOString(),
  });

  // Update an item with a composite key (partition key + sort key)
  await updateItemWithCompositeKey(
    "OrdersTable",
    "orderId",
    "order456",
    "productId",
    "prod789",
    "quantity",
    5
  );

  // Update with a condition
  await updateItemWithCondition(
    "UsersTable",
    "userId",
    "user123",
    "userStatus",
    "active",
    "userType",
    "premium"
  );

  // Batch update multiple items
  await batchUpdateItems("UsersTable", [
    {
      partitionKeyName: "userId",
      partitionKeyValue: "user123",
      attributeName: "lastLogin",
      attributeValue: new Date().toISOString(),
    },
    {
      partitionKeyName: "userId",
      partitionKeyValue: "user456",
      attributeName: "lastLogin",
      attributeValue: new Date().toISOString(),
    },
  ]);
};
```

- For API details, see the following topics in *AWS SDK for JavaScript API Reference*.

- BatchExecuteStatement
- ExecuteStatement

For a complete list of AWS SDK developer guides and code examples, see Using DynamoDB with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use API Gateway to invoke a Lambda function

The following code examples show how to create an AWS Lambda function invoked by Amazon API Gateway.

Java

SDK for Java 2.x

Shows how to create an AWS Lambda function by using the Lambda Java runtime API. This example invokes different AWS services to perform a specific use case. This example demonstrates how to create a Lambda function invoked by Amazon API Gateway that scans an Amazon DynamoDB table for work anniversaries and uses Amazon Simple Notification Service (Amazon SNS) to send a text message to your employees that congratulates them at their one year anniversary date.

For complete source code and instructions on how to set up and run, see the full example on GitHub.

Services used in this example

- API Gateway
- DynamoDB
- Lambda
- Amazon SNS

JavaScript

SDK for JavaScript (v3)

Shows how to create an AWS Lambda function by using the Lambda JavaScript runtime API. This example invokes different AWS services to perform a specific use case. This example

demonstrates how to create a Lambda function invoked by Amazon API Gateway that scans an Amazon DynamoDB table for work anniversaries and uses Amazon Simple Notification Service (Amazon SNS) to send a text message to your employees that congratulates them at their one year anniversary date.

For complete source code and instructions on how to set up and run, see the full example on GitHub.

This example is also available in the AWS SDK for JavaScript v3 developer guide.

Services used in this example

- API Gateway
- DynamoDB
- Lambda
- Amazon SNS

Python

SDK for Python (Boto3)

This example shows how to create and use an Amazon API Gateway REST API that targets an AWS Lambda function. The Lambda handler demonstrates how to route based on HTTP methods; how to get data from the query string, header, and body; and how to return a JSON response.

- Deploy a Lambda function.
- Create an API Gateway REST API.
- Create a REST resource that targets the Lambda function.
- Grant permission to let API Gateway invoke the Lambda function.
- Use the Requests package to send requests to the REST API.
- Clean up all resources created during the demo.

This example is best viewed on GitHub. For complete source code and instructions on how to set up and run, see the full example on GitHub.

Services used in this example

- API Gateway
- DynamoDB

- Lambda
- Amazon SNS

For a complete list of AWS SDK developer guides and code examples, see Using DynamoDB with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use Step Functions to invoke Lambda functions

The following code example shows how to create an AWS Step Functions state machine that invokes AWS Lambda functions in sequence.

Java

SDK for Java 2.x

Shows how to create an AWS serverless workflow by using AWS Step Functions and the AWS SDK for Java 2.x. Each workflow step is implemented using an AWS Lambda function.

For complete source code and instructions on how to set up and run, see the full example on GitHub.

Services used in this example

- DynamoDB
- Lambda
- Amazon SES
- Step Functions

For a complete list of AWS SDK developer guides and code examples, see Using DynamoDB with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use a document model for DynamoDB using an AWS SDK

The following code example shows how to perform Create, Read, Update, and Delete (CRUD) and batch operations using a document model for DynamoDB and an AWS SDK.

For more information, see Document model.

.NET

SDK for .NET

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Perform CRUD operations using a document model.

```
/// <summary>
/// Performs CRUD operations on an Amazon DynamoDB table.
/// </summary>
public class MidlevelItemCRUD
{
    public static async Task Main()
    {
        var tableName = "ProductCatalog";
        var sampleBookId = 555;

        var client = new AmazonDynamoDBClient();
        var productCatalog = LoadTable(client, tableName);

        await CreateBookItem(productCatalog, sampleBookId);
        RetrieveBook(productCatalog, sampleBookId);

        // Couple of sample updates.
        UpdateMultipleAttributes(productCatalog, sampleBookId);
        UpdateBookPriceConditionally(productCatalog, sampleBookId);

        // Delete.
        await DeleteBook(productCatalog, sampleBookId);
    }

    /// <summary>
    /// Loads the contents of a DynamoDB table.
    /// </summary>
    /// <param name="client">An initialized DynamoDB client object.</param>
    /// <param name="tableName">The name of the table to load.</param>
    /// <returns>A DynamoDB table object.</returns>
```

```
        public static Table LoadTable(IAmazonDynamoDB client, string tableName)
        {
            Table productCatalog = Table.LoadTable(client, tableName);
            return productCatalog;
        }

        /// <summary>
        /// Creates an example book item and adds it to the DynamoDB table
        /// ProductCatalog.
        /// </summary>
        /// <param name="productCatalog">A DynamoDB table object.</param>
        /// <param name="sampleBookId">An integer value representing the book's
ID.</param>
        public static async Task CreateBookItem(Table productCatalog, int
sampleBookId)
        {
            Console.WriteLine("\n*** Executing CreateBookItem() ***");
            var book = new Document
            {
                ["Id"] = sampleBookId,
                ["Title"] = "Book " + sampleBookId,
                ["Price"] = 19.99,
                ["ISBN"] = "111-1111111111",
                ["Authors"] = new List<string> { "Author 1", "Author 2", "Author
3" },
                ["PageCount"] = 500,
                ["Dimensions"] = "8.5x11x.5",
                ["InPublication"] = new DynamoDBBool(true),
                ["InStock"] = new DynamoDBBool(false),
                ["QuantityOnHand"] = 0,
            };

            // Adds the book to the ProductCatalog table.
            await productCatalog.PutItemAsync(book);
        }

        /// <summary>
        /// Retrieves an item, a book, from the DynamoDB ProductCatalog table.
        /// </summary>
        /// <param name="productCatalog">A DynamoDB table object.</param>
        /// <param name="sampleBookId">An integer value representing the book's
ID.</param>
        public static async void RetrieveBook(
          Table productCatalog,
```

```
          int sampleBookId)
        {
            Console.WriteLine("\n*** Executing RetrieveBook() ***");

            // Optional configuration.
            var config = new GetItemOperationConfig
            {
                AttributesToGet = new List<string> { "Id", "ISBN", "Title",
"Authors", "Price" },
                ConsistentRead = true,
            };

            Document document = await productCatalog.GetItemAsync(sampleBookId,
config);
            Console.WriteLine("RetrieveBook: Printing book retrieved...");
            PrintDocument(document);
        }

        /// <summary>
        /// Updates multiple attributes for a book and writes the changes to the
        /// DynamoDB table ProductCatalog.
        /// </summary>
        /// <param name="productCatalog">A DynamoDB table object.</param>
        /// <param name="sampleBookId">An integer value representing the book's
ID.</param>
        public static async void UpdateMultipleAttributes(
          Table productCatalog,
          int sampleBookId)
        {
            Console.WriteLine("\nUpdating multiple attributes....");
            int partitionKey = sampleBookId;

            var book = new Document
            {
                ["Id"] = partitionKey,

                // List of attribute updates.
                // The following replaces the existing authors list.
                ["Authors"] = new List<string> { "Author x", "Author y" },
                ["newAttribute"] = "New Value",
                ["ISBN"] = null, // Remove it.
            };

            // Optional parameters.
```

```
            var config = new UpdateItemOperationConfig
            {
                // Gets updated item in response.
                ReturnValues = ReturnValues.AllNewAttributes,
            };

            Document updatedBook = await productCatalog.UpdateItemAsync(book,
config);
            Console.WriteLine("UpdateMultipleAttributes: Printing item after
updates ...");
            PrintDocument(updatedBook);
        }

        /// <summary>
        /// Updates a book item if it meets the specified criteria.
        /// </summary>
        /// <param name="productCatalog">A DynamoDB table object.</param>
        /// <param name="sampleBookId">An integer value representing the book's
ID.</param>
        public static async void UpdateBookPriceConditionally(
          Table productCatalog,
          int sampleBookId)
        {
            Console.WriteLine("\n*** Executing UpdateBookPriceConditionally()
***");

            int partitionKey = sampleBookId;

            var book = new Document
            {
                ["Id"] = partitionKey,
                ["Price"] = 29.99,
            };

            // For conditional price update, creating a condition expression.
            var expr = new Expression
            {
                ExpressionStatement = "Price = :val",
            };
            expr.ExpressionAttributeValues[":val"] = 19.00;

            // Optional parameters.
            var config = new UpdateItemOperationConfig
            {
```

```
                ConditionalExpression = expr,
                ReturnValues = ReturnValues.AllNewAttributes,
            };

            Document updatedBook = await productCatalog.UpdateItemAsync(book,
config);
            Console.WriteLine("UpdateBookPriceConditionally: Printing item whose
price was conditionally updated");
            PrintDocument(updatedBook);
        }

        /// <summary>
        /// Deletes the book with the supplied Id value from the DynamoDB table
        /// ProductCatalog.
        /// </summary>
        /// <param name="productCatalog">A DynamoDB table object.</param>
        /// <param name="sampleBookId">An integer value representing the book's
ID.</param>
        public static async Task DeleteBook(
          Table productCatalog,
          int sampleBookId)
        {
            Console.WriteLine("\n*** Executing DeleteBook() ***");

            // Optional configuration.
            var config = new DeleteItemOperationConfig
            {
                // Returns the deleted item.
                ReturnValues = ReturnValues.AllOldAttributes,
            };
            Document document = await
productCatalog.DeleteItemAsync(sampleBookId, config);
            Console.WriteLine("DeleteBook: Printing deleted just deleted...");

            PrintDocument(document);
        }

        /// <summary>
        /// Prints the information for the supplied DynamoDB document.
        /// </summary>
        /// <param name="updatedDocument">A DynamoDB document object.</param>
        public static void PrintDocument(Document updatedDocument)
        {
            if (updatedDocument is null)
```

```
            {
                return;
            }

            foreach (var attribute in updatedDocument.GetAttributeNames())
            {
                string stringValue = null;
                var value = updatedDocument[attribute];

                if (value is null)
                {
                    continue;
                }

                if (value is Primitive)
                {
                    stringValue = value.AsPrimitive().Value.ToString();
                }
                else if (value is PrimitiveList)
                {
                    stringValue = string.Join(",", (from primitive
                      in value.AsPrimitiveList().Entries
                                                    select
primitive.Value).ToArray());
                }

                Console.WriteLine($"{attribute} - {stringValue}", attribute,
stringValue);
            }
        }
    }
```

Perform batch write operations using a document model.

```
    /// <summary>
    /// Shows how to use mid-level Amazon DynamoDB API calls to perform batch
    /// operations.
    /// </summary>
    public class MidLevelBatchWriteItem
    {
```

```
        public static async Task Main()
        {
            IAmazonDynamoDB client = new AmazonDynamoDBClient();

            await SingleTableBatchWrite(client);
            await MultiTableBatchWrite(client);
        }

        /// <summary>
        /// Perform a batch operation on a single DynamoDB table.
        /// </summary>
        /// <param name="client">An initialized DynamoDB object.</param>
        public static async Task SingleTableBatchWrite(IAmazonDynamoDB client)
        {
            Table productCatalog = Table.LoadTable(client, "ProductCatalog");
            var batchWrite = productCatalog.CreateBatchWrite();

            var book1 = new Document
            {
                ["Id"] = 902,
                ["Title"] = "My book1 in batch write using .NET helper classes",
                ["ISBN"] = "902-11-11-1111",
                ["Price"] = 10,
                ["ProductCategory"] = "Book",
                ["Authors"] = new List<string> { "Author 1", "Author 2", "Author
3" },
                ["Dimensions"] = "8.5x11x.5",
                ["InStock"] = new DynamoDBBool(true),
                ["QuantityOnHand"] = new DynamoDBNull(), // Quantity is unknown
at this time.
            };

            batchWrite.AddDocumentToPut(book1);

            // Specify delete item using overload that takes PK.
            batchWrite.AddKeyToDelete(12345);
            Console.WriteLine("Performing batch write in
SingleTableBatchWrite()");
            await batchWrite.ExecuteAsync();
        }

        /// <summary>
        /// Perform a batch operation involving multiple DynamoDB tables.
        /// </summary>
```

```
        /// <param name="client">An initialized DynamoDB client object.</param>
        public static async Task MultiTableBatchWrite(IAmazonDynamoDB client)
        {
            // Specify item to add in the Forum table.
            Table forum = Table.LoadTable(client, "Forum");
            var forumBatchWrite = forum.CreateBatchWrite();

            var forum1 = new Document
            {
                ["Name"] = "Test BatchWrite Forum",
                ["Threads"] = 0,
            };
            forumBatchWrite.AddDocumentToPut(forum1);

            // Specify item to add in the Thread table.
            Table thread = Table.LoadTable(client, "Thread");
            var threadBatchWrite = thread.CreateBatchWrite();

            var thread1 = new Document
            {
                ["ForumName"] = "S3 forum",
                ["Subject"] = "My sample question",
                ["Message"] = "Message text",
                ["KeywordTags"] = new List<string> { "S3", "Bucket" },
            };
            threadBatchWrite.AddDocumentToPut(thread1);

            // Specify item to delete from the Thread table.
            threadBatchWrite.AddKeyToDelete("someForumName", "someSubject");

            // Create multi-table batch.
            var superBatch = new MultiTableDocumentBatchWrite();
            superBatch.AddBatch(forumBatchWrite);
            superBatch.AddBatch(threadBatchWrite);
            Console.WriteLine("Performing batch write in
MultiTableBatchWrite()");

            // Execute the batch.
            await superBatch.ExecuteAsync();
        }
    }
```

Scan a table using a document model.

```
   /// <summary>
   /// Shows how to use mid-level Amazon DynamoDB API calls to scan a DynamoDB
   /// table for values.
   /// </summary>
   public class MidLevelScanOnly
   {
       public static async Task Main()
       {
           IAmazonDynamoDB client = new AmazonDynamoDBClient();

           Table productCatalogTable = Table.LoadTable(client,
"ProductCatalog");

           await FindProductsWithNegativePrice(productCatalogTable);
           await FindProductsWithNegativePriceWithConfig(productCatalogTable);
       }

       /// <summary>
       /// Retrieves any products that have a negative price in a DynamoDB
table.
       /// </summary>
       /// <param name="productCatalogTable">A DynamoDB table object.</param>
       public static async Task FindProductsWithNegativePrice(
         Table productCatalogTable)
       {
           // Assume there is a price error. So we scan to find items priced <
0.
           var scanFilter = new ScanFilter();
           scanFilter.AddCondition("Price", ScanOperator.LessThan, 0);

           Search search = productCatalogTable.Scan(scanFilter);

           do
           {
               var documentList = await search.GetNextSetAsync();
               Console.WriteLine("\nFindProductsWithNegativePrice:
printing .....");

               foreach (var document in documentList)
               {
                   PrintDocument(document);
```

```
                }
            }
            while (!search.IsDone);
        }

        /// <summary>
        /// Finds any items in the ProductCatalog table using a DynamoDB
        /// configuration object.
        /// </summary>
        /// <param name="productCatalogTable">A DynamoDB table object.</param>
        public static async Task FindProductsWithNegativePriceWithConfig(
          Table productCatalogTable)
        {
            // Assume there is a price error. So we scan to find items priced <
0.
            var scanFilter = new ScanFilter();
            scanFilter.AddCondition("Price", ScanOperator.LessThan, 0);

            var config = new ScanOperationConfig()
            {
                Filter = scanFilter,
                Select = SelectValues.SpecificAttributes,
                AttributesToGet = new List<string> { "Title", "Id" },
            };

            Search search = productCatalogTable.Scan(config);

            do
            {
                var documentList = await search.GetNextSetAsync();
                Console.WriteLine("\nFindProductsWithNegativePriceWithConfig:
printing .....");

                foreach (var document in documentList)
                {
                    PrintDocument(document);
                }
            }
            while (!search.IsDone);
        }

        /// <summary>
        /// Displays the details of the passed DynamoDB document object on the
        /// console.
```

```
        /// </summary>
        /// <param name="document">A DynamoDB document object.</param>
        public static void PrintDocument(Document document)
        {
            Console.WriteLine();
            foreach (var attribute in document.GetAttributeNames())
            {
                string stringValue = null;
                var value = document[attribute];
                if (value is Primitive)
                {
                    stringValue = value.AsPrimitive().Value.ToString();
                }
                else if (value is PrimitiveList)
                {
                    stringValue = string.Join(",", (from primitive
                      in value.AsPrimitiveList().Entries
                                                    select
primitive.Value).ToArray());
                }

                Console.WriteLine($"{attribute} - {stringValue}");
            }
        }
    }
```

Query and scan a table using a document model.

```
    /// <summary>
    /// Shows how to perform mid-level query procedures on an Amazon DynamoDB
    /// table.
    /// </summary>
    public class MidLevelQueryAndScan
    {
        public static async Task Main()
        {
            IAmazonDynamoDB client = new AmazonDynamoDBClient();

            // Query examples.
            Table replyTable = Table.LoadTable(client, "Reply");
```

```
            string forumName = "Amazon DynamoDB";
            string threadSubject = "DynamoDB Thread 2";

            await FindRepliesInLast15Days(replyTable);
            await FindRepliesInLast15DaysWithConfig(replyTable, forumName,
threadSubject);
            await FindRepliesPostedWithinTimePeriod(replyTable, forumName,
threadSubject);

            // Get Example.
            Table productCatalogTable = Table.LoadTable(client,
"ProductCatalog");
            int productId = 101;

            await GetProduct(productCatalogTable, productId);
        }

        /// <summary>
        /// Retrieves information about a product from the DynamoDB table
        /// ProductCatalog based on the product ID and displays the information
        /// on the console.
        /// </summary>
        /// <param name="tableName">The name of the table from which to retrieve
        /// product information.</param>
        /// <param name="productId">The ID of the product to retrieve.</param>
        public static async Task GetProduct(Table tableName, int productId)
        {
            Console.WriteLine("*** Executing GetProduct() ***");
            Document productDocument = await tableName.GetItemAsync(productId);
            if (productDocument != null)
            {
                PrintDocument(productDocument);
            }
            else
            {
                Console.WriteLine("Error: product " + productId + " does not
exist");
            }
        }

        /// <summary>
        /// Retrieves replies from the passed DynamoDB table object.
        /// </summary>
        /// <param name="table">The table we want to query.</param>
```

```
        public static async Task FindRepliesInLast15Days(
          Table table)
        {
            DateTime twoWeeksAgoDate = DateTime.UtcNow - TimeSpan.FromDays(15);
            var filter = new QueryFilter("Id", QueryOperator.Equal, "Id");
            filter.AddCondition("ReplyDateTime", QueryOperator.GreaterThan,
twoWeeksAgoDate);

            // Use Query overloads that take the minimum required query
parameters.
            Search search = table.Query(filter);

            do
            {
                var documentSet = await search.GetNextSetAsync();
                Console.WriteLine("\nFindRepliesInLast15Days:
printing .............");

                foreach (var document in documentSet)
                {
                    PrintDocument(document);
                }
            }
            while (!search.IsDone);
        }

        /// <summary>
        /// Retrieve replies made during a specific time period.
        /// </summary>
        /// <param name="table">The table we want to query.</param>
        /// <param name="forumName">The name of the forum that we're interested
in.</param>
        /// <param name="threadSubject">The subject of the thread, which we are
        /// searching for replies.</param>
        public static async Task FindRepliesPostedWithinTimePeriod(
          Table table,
          string forumName,
          string threadSubject)
        {
            DateTime startDate = DateTime.UtcNow.Subtract(new TimeSpan(21, 0, 0,
0));
            DateTime endDate = DateTime.UtcNow.Subtract(new TimeSpan(1, 0, 0,
0));
```

```
            var filter = new QueryFilter("Id", QueryOperator.Equal, forumName +
"#" + threadSubject);
            filter.AddCondition("ReplyDateTime", QueryOperator.Between,
startDate, endDate);

            var config = new QueryOperationConfig()
            {
                Limit = 2, // 2 items/page.
                Select = SelectValues.SpecificAttributes,
                AttributesToGet = new List<string>
       {
         "Message",
         "ReplyDateTime",
         "PostedBy",
       },
                ConsistentRead = true,
                Filter = filter,
            };

            Search search = table.Query(config);

            do
            {
                var documentList = await search.GetNextSetAsync();
                Console.WriteLine("\nFindRepliesPostedWithinTimePeriod: printing
replies posted within dates: {0} and {1} .....", startDate, endDate);

                foreach (var document in documentList)
                {
                    PrintDocument(document);
                }
            }
            while (!search.IsDone);
       }

       /// <summary>
       /// Perform a query for replies made in the last 15 days using a DynamoDB
       /// QueryOperationConfig object.
       /// </summary>
       /// <param name="table">The table we want to query.</param>
       /// <param name="forumName">The name of the forum that we're interested
in.</param>
       /// <param name="threadName">The bane of the thread that we are searching
       /// for replies.</param>
```

```
        public static async Task FindRepliesInLast15DaysWithConfig(
          Table table,
          string forumName,
          string threadName)
        {
            DateTime twoWeeksAgoDate = DateTime.UtcNow - TimeSpan.FromDays(15);
            var filter = new QueryFilter("Id", QueryOperator.Equal, forumName +
"#" + threadName);
            filter.AddCondition("ReplyDateTime", QueryOperator.GreaterThan,
twoWeeksAgoDate);

            var config = new QueryOperationConfig()
            {
                Filter = filter,

                // Optional parameters.
                Select = SelectValues.SpecificAttributes,
                AttributesToGet = new List<string>
                {
                  "Message",
                  "ReplyDateTime",
                  "PostedBy",
                },
                ConsistentRead = true,
            };

            Search search = table.Query(config);

            do
            {
                var documentSet = await search.GetNextSetAsync();
                Console.WriteLine("\nFindRepliesInLast15DaysWithConfig:
printing .....");

                foreach (var document in documentSet)
                {
                    PrintDocument(document);
                }
            }
            while (!search.IsDone);
        }

        /// <summary>
        /// Displays the contents of the passed DynamoDB document on the console.
```

```
        /// </summary>
        /// <param name="document">A DynamoDB document to display.</param>
        public static void PrintDocument(Document document)
        {
            Console.WriteLine();
            foreach (var attribute in document.GetAttributeNames())
            {
                string stringValue = null;
                var value = document[attribute];

                if (value is Primitive)
                {
                    stringValue = value.AsPrimitive().Value.ToString();
                }
                else if (value is PrimitiveList)
                {
                    stringValue = string.Join(",", (from primitive
                      in value.AsPrimitiveList().Entries
                                                    select
primitive.Value).ToArray());
                }

                Console.WriteLine($"{attribute} - {stringValue}");
            }
        }
    }
```

For a complete list of AWS SDK developer guides and code examples, see Using DynamoDB with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use a high-level object persistence model for DynamoDB using an AWS SDK

The following code example shows how to perform Create, Read, Update, and Delete (CRUD) and batch operations using an object persistence model for DynamoDB and an AWS SDK.

For more information, see Object persistence model.

.NET

SDK for .NET

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Perform CRUD operations using a high-level object persistence model.

```
/// <summary>
/// Shows how to perform high-level CRUD operations on an Amazon DynamoDB
/// table.
/// </summary>
public class HighLevelItemCrud
{
    public static async Task Main()
    {
        var client = new AmazonDynamoDBClient();
        DynamoDBContext context = new DynamoDBContext(client);
        await PerformCRUDOperations(context);
    }

    public static async Task PerformCRUDOperations(IDynamoDBContext context)
    {
        int bookId = 1001; // Some unique value.
        Book myBook = new Book
        {
            Id = bookId,
            Title = "object persistence-AWS SDK for.NET SDK-Book 1001",
            Isbn = "111-1111111001",
            BookAuthors = new List<string> { "Author 1", "Author 2" },
        };

        // Save the book to the ProductCatalog table.
        await context.SaveAsync(myBook);

        // Retrieve the book from the ProductCatalog table.
        Book bookRetrieved = await context.LoadAsync<Book>(bookId);
```

```
            // Update some properties.
            bookRetrieved.Isbn = "222-2222221001";

            // Update existing authors list with the following values.
            bookRetrieved.BookAuthors = new List<string> { " Author 1", "Author
x" };
            await context.SaveAsync(bookRetrieved);

            // Retrieve the updated book. This time, add the optional
            // ConsistentRead parameter using DynamoDBContextConfig object.
            await context.LoadAsync<Book>(bookId, new DynamoDBContextConfig
            {
                ConsistentRead = true,
            });

            // Delete the book.
            await context.DeleteAsync<Book>(bookId);

            // Try to retrieve deleted book. It should return null.
            Book deletedBook = await context.LoadAsync<Book>(bookId, new
DynamoDBContextConfig
            {
                ConsistentRead = true,
            });

            if (deletedBook == null)
            {
                Console.WriteLine("Book is deleted");
            }
        }
    }
```

Perform batch write operations using a high-level object persistence model.

```
    /// <summary>
    /// Performs high-level batch write operations to an Amazon DynamoDB table.
    /// This example was written using the AWS SDK for .NET version 3.7 and .NET
    /// Core 5.0.
    /// </summary>
    public class HighLevelBatchWriteItem
```

```
    {
        public static async Task SingleTableBatchWrite(IDynamoDBContext context)
        {
            Book book1 = new Book
            {
                Id = 902,
                InPublication = true,
                Isbn = "902-11-11-1111",
                PageCount = "100",
                Price = 10,
                ProductCategory = "Book",
                Title = "My book3 in batch write",
            };

            Book book2 = new Book
            {
                Id = 903,
                InPublication = true,
                Isbn = "903-11-11-1111",
                PageCount = "200",
                Price = 10,
                ProductCategory = "Book",
                Title = "My book4 in batch write",
            };

            var bookBatch = context.CreateBatchWrite<Book>();
            bookBatch.AddPutItems(new List<Book> { book1, book2 });

            Console.WriteLine("Adding two books to ProductCatalog table.");
            await bookBatch.ExecuteAsync();
        }

        public static async Task MultiTableBatchWrite(IDynamoDBContext context)
        {
            // New Forum item.
            Forum newForum = new Forum
            {
                Name = "Test BatchWrite Forum",
                Threads = 0,
            };
            var forumBatch = context.CreateBatchWrite<Forum>();
            forumBatch.AddPutItem(newForum);

            // New Thread item.
```

```
            Thread newThread = new Thread
            {
                ForumName = "S3 forum",
                Subject = "My sample question",
                KeywordTags = new List<string> { "S3", "Bucket" },
                Message = "Message text",
            };

            DynamoDBOperationConfig config = new DynamoDBOperationConfig();
            config.SkipVersionCheck = true;
            var threadBatch = context.CreateBatchWrite<Thread>(config);
            threadBatch.AddPutItem(newThread);
            threadBatch.AddDeleteKey("some partition key value", "some sort key
value");

            var superBatch = new MultiTableBatchWrite(forumBatch, threadBatch);

            Console.WriteLine("Performing batch write in
MultiTableBatchWrite().");
            await superBatch.ExecuteAsync();
        }

        public static async Task Main()
        {
            AmazonDynamoDBClient client = new AmazonDynamoDBClient();
            DynamoDBContext context = new DynamoDBContext(client);

            await SingleTableBatchWrite(context);
            await MultiTableBatchWrite(context);
        }
    }
```

Map arbitrary data to a table using a high-level object persistence model.

```
    /// <summary>
    /// Shows how to map arbitrary data to an Amazon DynamoDB table.
    /// </summary>
    public class HighLevelMappingArbitraryData
    {
        /// <summary>
```

```
        /// Creates a book, adds it to the DynamoDB ProductCatalog table,
retrieves
        /// the new book from the table, updates the dimensions and writes the
        /// changed item back to the table.
        /// </summary>
        /// <param name="context">The DynamoDB context object used to write and
        /// read data from the table.</param>
        public static async Task AddRetrieveUpdateBook(IDynamoDBContext context)
        {
            // Create a book.
            DimensionType myBookDimensions = new DimensionType()
            {
                Length = 8M,
                Height = 11M,
                Thickness = 0.5M,
            };

            Book myBook = new Book
            {
                Id = 501,
                Title = "AWS SDK for .NET Object Persistence Model Handling
Arbitrary Data",
                Isbn = "999-9999999999",
                BookAuthors = new List<string> { "Author 1", "Author 2" },
                Dimensions = myBookDimensions,
            };

            // Add the book to the DynamoDB table ProductCatalog.
            await context.SaveAsync(myBook);

            // Retrieve the book.
            Book bookRetrieved = await context.LoadAsync<Book>(501);

            // Update the book dimensions property.
            bookRetrieved.Dimensions.Height += 1;
            bookRetrieved.Dimensions.Length += 1;
            bookRetrieved.Dimensions.Thickness += 0.2M;

            // Write the changed item to the table.
            await context.SaveAsync(bookRetrieved);
        }

        public static async Task Main()
        {
```

```
            var client = new AmazonDynamoDBClient();
            DynamoDBContext context = new DynamoDBContext(client);
            await AddRetrieveUpdateBook(context);
        }
    }
```

Query and scan a table using a high-level object persistence model.

```
    /// <summary>
    /// Shows how to perform high-level query and scan operations to Amazon
    /// DynamoDB tables.
    /// </summary>
    public class HighLevelQueryAndScan
    {
        public static async Task Main()
        {
            var client = new AmazonDynamoDBClient();

            DynamoDBContext context = new DynamoDBContext(client);

            // Get an item.
            await GetBook(context, 101);

            // Sample forum and thread to test queries.
            string forumName = "Amazon DynamoDB";
            string threadSubject = "DynamoDB Thread 1";

            // Sample queries.
            await FindRepliesInLast15Days(context, forumName, threadSubject);
            await FindRepliesPostedWithinTimePeriod(context, forumName,
threadSubject);

            // Scan table.
            await FindProductsPricedLessThanZero(context);
        }

        public static async Task GetBook(IDynamoDBContext context, int productId)
        {
            Book bookItem = await context.LoadAsync<Book>(productId);
```

```
            Console.WriteLine("\nGetBook: Printing result.....");
            Console.WriteLine($"Title: {bookItem.Title} \n ISBN:{bookItem.Isbn}
\n No. of pages: {bookItem.PageCount}");
        }

        /// <summary>
        /// Queries a DynamoDB table to find replies posted within the last 15
days.
        /// </summary>
        /// <param name="context">The DynamoDB context used to perform the
query.</param>
        /// <param name="forumName">The name of the forum that we're interested
in.</param>
        /// <param name="threadSubject">The thread object containing the query
parameters.</param>
        public static async Task FindRepliesInLast15Days(
          IDynamoDBContext context,
          string forumName,
          string threadSubject)
        {
            string replyId = $"{forumName} #{threadSubject}";
            DateTime twoWeeksAgoDate = DateTime.UtcNow - TimeSpan.FromDays(15);

            List<object> times = new List<object>();
            times.Add(twoWeeksAgoDate);

            List<ScanCondition> scs = new List<ScanCondition>();
            var sc = new ScanCondition("PostedBy", ScanOperator.GreaterThan,
times.ToArray());
            scs.Add(sc);

            var cfg = new DynamoDBOperationConfig
            {
                QueryFilter = scs,
            };

            AsyncSearch<Reply> response = context.QueryAsync<Reply>(replyId,
cfg);
            IEnumerable<Reply> latestReplies = await
response.GetRemainingAsync();

            Console.WriteLine("\nReplies in last 15 days:");

            foreach (Reply r in latestReplies)
```

```
            {

Console.WriteLine($"{r.Id}\t{r.PostedBy}\t{r.Message}\t{r.ReplyDateTime}");
            }
        }

        /// <summary>
        /// Queries for replies posted within a specific time period.
        /// </summary>
        /// <param name="context">The DynamoDB context used to perform the
query.</param>
        /// <param name="forumName">The name of the forum that we're interested
in.</param>
        /// <param name="threadSubject">Information about the subject that we're
        /// interested in.</param>
        public static async Task FindRepliesPostedWithinTimePeriod(
          IDynamoDBContext context,
          string forumName,
          string threadSubject)
        {
            string forumId = forumName + "#" + threadSubject;
            Console.WriteLine("\nReplies posted within time period:");

            DateTime startDate = DateTime.UtcNow - TimeSpan.FromDays(30);
            DateTime endDate = DateTime.UtcNow - TimeSpan.FromDays(1);

            List<object> times = new List<object>();
            times.Add(startDate);
            times.Add(endDate);

            List<ScanCondition> scs = new List<ScanCondition>();
            var sc = new ScanCondition("LastPostedBy", ScanOperator.Between,
times.ToArray());
            scs.Add(sc);

            var cfg = new DynamoDBOperationConfig
            {
                QueryFilter = scs,
            };

            AsyncSearch<Reply> response = context.QueryAsync<Reply>(forumId,
cfg);
            IEnumerable<Reply> repliesInAPeriod = await
response.GetRemainingAsync();
```

```
            foreach (Reply r in repliesInAPeriod)
            {

Console.WriteLine("{r.Id}\t{r.PostedBy}\t{r.Message}\t{r.ReplyDateTime}");
            }
        }

        /// <summary>
        /// Queries the DynamoDB ProductCatalog table for products costing less
        /// than zero.
        /// </summary>
        /// <param name="context">The DynamoDB context object used to perform the
        /// query.</param>
        public static async Task FindProductsPricedLessThanZero(IDynamoDBContext
context)
        {
            int price = 0;

            List<ScanCondition> scs = new List<ScanCondition>();
            var sc1 = new ScanCondition("Price", ScanOperator.LessThan, price);
            var sc2 = new ScanCondition("ProductCategory", ScanOperator.Equal,
"Book");
            scs.Add(sc1);
            scs.Add(sc2);

            AsyncSearch<Book> response = context.ScanAsync<Book>(scs);

            IEnumerable<Book> itemsWithWrongPrice = await
response.GetRemainingAsync();

            Console.WriteLine("\nFindProductsPricedLessThanZero: Printing
result.....");

            foreach (Book r in itemsWithWrongPrice)
            {
                Console.WriteLine($"{r.Id}\t{r.Title}\t{r.Price}\t{r.Isbn}");
            }
        }
    }
```

For a complete list of AWS SDK developer guides and code examples, see Using DynamoDB with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use atomic counter operations in DynamoDB with an AWS SDK

The following code examples show how to use atomic counter operations in DynamoDB.

- Increment counters atomically using ADD and SET operations.
- Safely increment counters that might not exist.
- Implement optimistic locking for counter operations.

Java

SDK for Java 2.x

Demonstrate atomic counter operations using AWS SDK for Java 2.x.

```
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.GetItemRequest;
import software.amazon.awssdk.services.dynamodb.model.GetItemResponse;
import software.amazon.awssdk.services.dynamodb.model.ReturnValue;
import software.amazon.awssdk.services.dynamodb.model.UpdateItemRequest;
import software.amazon.awssdk.services.dynamodb.model.UpdateItemResponse;

import java.util.HashMap;
import java.util.Map;

    /**
     * Increments a counter using the ADD operation.
     *
     * <p>This method demonstrates how to use the ADD operation to atomically
     * increment a counter attribute.
     *
     * @param dynamoDbClient The DynamoDB client
     * @param tableName The name of the DynamoDB table
     * @param key The key of the item to update
     * @param counterName The name of the counter attribute
     * @param incrementValue The value to increment by
     * @return The response from DynamoDB
```

```
     * @throws DynamoDbException if an error occurs during the operation
     */
    public static UpdateItemResponse incrementCounterWithAdd(
        DynamoDbClient dynamoDbClient,
        String tableName,
        Map<String, AttributeValue> key,
        String counterName,
        int incrementValue) {

        // Define the update parameters
        UpdateItemRequest request = UpdateItemRequest.builder()
            .tableName(tableName)
            .key(key)
            .updateExpression("ADD #counterName :increment")
            .expressionAttributeNames(Map.of("#counterName", counterName))
            .expressionAttributeValues(Map.of(
                ":increment",

AttributeValue.builder().n(String.valueOf(incrementValue)).build()))
            .returnValues(ReturnValue.UPDATED_NEW)
            .build();

        // Perform the update operation
        return dynamoDbClient.updateItem(request);
    }

    /**
     * Increments a counter using the SET operation.
     *
     * <p>This method demonstrates how to use the SET operation with an
expression
     * to increment a counter attribute.
     *
     * @param dynamoDbClient The DynamoDB client
     * @param tableName The name of the DynamoDB table
     * @param key The key of the item to update
     * @param counterName The name of the counter attribute
     * @param incrementValue The value to increment by
     * @return The response from DynamoDB
     * @throws DynamoDbException if an error occurs during the operation
     */
    public static UpdateItemResponse incrementCounterWithSet(
        DynamoDbClient dynamoDbClient,
        String tableName,
```

```
        Map<String, AttributeValue> key,
        String counterName,
        int incrementValue) {

        // Define the update parameters
        UpdateItemRequest request = UpdateItemRequest.builder()
            .tableName(tableName)
            .key(key)
            .updateExpression("SET #counterName = #counterName + :increment")
            .expressionAttributeNames(Map.of("#counterName", counterName))
            .expressionAttributeValues(Map.of(
                ":increment",

AttributeValue.builder().n(String.valueOf(incrementValue)).build()))
            .returnValues(ReturnValue.UPDATED_NEW)
            .build();

        // Perform the update operation
        return dynamoDbClient.updateItem(request);
    }

    /**
     * Increments a counter safely, handling the case where the counter doesn't
exist yet.
     *
     * <p>This method demonstrates how to use if_not_exists to safely increment a
counter
     * that may not exist yet.
     *
     * @param dynamoDbClient The DynamoDB client
     * @param tableName The name of the DynamoDB table
     * @param key The key of the item to update
     * @param counterName The name of the counter attribute
     * @param incrementValue The value to increment by
     * @return The response from DynamoDB
     * @throws DynamoDbException if an error occurs during the operation
     */
    public static UpdateItemResponse incrementCounterSafely(
        DynamoDbClient dynamoDbClient,
        String tableName,
        Map<String, AttributeValue> key,
        String counterName,
        int incrementValue) {
```

```
        // Define the update parameters
        UpdateItemRequest request = UpdateItemRequest.builder()
            .tableName(tableName)
            .key(key)
            .updateExpression("SET #counterName =
if_not_exists(#counterName, :zero) + :increment")
            .expressionAttributeNames(Map.of("#counterName", counterName))
            .expressionAttributeValues(Map.of(
                ":increment",

AttributeValue.builder().n(String.valueOf(incrementValue)).build(),
                ":zero", AttributeValue.builder().n("0").build()))
            .returnValues(ReturnValue.UPDATED_NEW)
            .build();

        // Perform the update operation
        return dynamoDbClient.updateItem(request);
    }

    /**
     * Decrements a counter safely, ensuring it doesn't go below zero.
     *
     * <p>This method demonstrates how to use a condition expression to safely
     * decrement a counter without going below zero.
     *
     * @param dynamoDbClient The DynamoDB client
     * @param tableName The name of the DynamoDB table
     * @param key The key of the item to update
     * @param counterName The name of the counter attribute
     * @param decrementValue The value to decrement by
     * @return The response from DynamoDB
     * @throws DynamoDbException if an error occurs during the operation or if
the counter would go below zero
     */
    public static UpdateItemResponse decrementCounterSafely(
        DynamoDbClient dynamoDbClient,
        String tableName,
        Map<String, AttributeValue> key,
        String counterName,
        int decrementValue) {

        // Define the update parameters
        UpdateItemRequest request = UpdateItemRequest.builder()
            .tableName(tableName)
```

```
            .key(key)
            .updateExpression("SET #counterName = #counterName - :decrement")
            .conditionExpression("#counterName >= :decrement")
            .expressionAttributeNames(Map.of("#counterName", counterName))
            .expressionAttributeValues(Map.of(
                ":decrement",

AttributeValue.builder().n(String.valueOf(decrementValue)).build()))
            .returnValues(ReturnValue.UPDATED_NEW)
            .build();

        // Perform the update operation
        return dynamoDbClient.updateItem(request);
    }

    /**
     * Compares the ADD and SET approaches for incrementing counters.
     *
     * <p>This method demonstrates the differences between using ADD and SET
     * for incrementing counters in DynamoDB.
     *
     * @param dynamoDbClient The DynamoDB client
     * @param tableName The name of the DynamoDB table
     * @param key The key of the item to update
     * @return Map containing the comparison results
     */
    public static Map<String, Object> compareAddVsSet(
        DynamoDbClient dynamoDbClient, String tableName, Map<String,
AttributeValue> key) {

        Map<String, Object> results = new HashMap<>();

        try {
            // Reset counters to ensure a fair comparison
            UpdateItemRequest resetRequest = UpdateItemRequest.builder()
                .tableName(tableName)
                .key(key)
                .updateExpression("SET AddCounter = :zero, SetCounter = :zero")
                .expressionAttributeValues(
                    Map.of(":zero", AttributeValue.builder().n("0").build()))
                .build();

            dynamoDbClient.updateItem(resetRequest);
```

```
            // Increment with ADD
            long addStartTime = System.nanoTime();
            UpdateItemResponse addResponse =
incrementCounterWithAdd(dynamoDbClient, tableName, key, "AddCounter", 1);
            long addEndTime = System.nanoTime();
            long addDuration = addEndTime - addStartTime;

            // Increment with SET
            long setStartTime = System.nanoTime();
            UpdateItemResponse setResponse =
incrementCounterWithSet(dynamoDbClient, tableName, key, "SetCounter", 1);
            long setEndTime = System.nanoTime();
            long setDuration = setEndTime - setStartTime;

            // Record results
            results.put("addResponse", addResponse);
            results.put("setResponse", setResponse);
            results.put("addDuration", addDuration);
            results.put("setDuration", setDuration);
            results.put("success", true);

        } catch (DynamoDbException e) {
            results.put("success", false);
            results.put("error", e.getMessage());
        }

        return results;
    }

    /**
     * Gets the current value of a counter attribute.
     *
     * <p>Helper method to retrieve the current value of a counter attribute.
     *
     * @param dynamoDbClient The DynamoDB client
     * @param tableName The name of the DynamoDB table
     * @param key The key of the item to get
     * @param counterName The name of the counter attribute
     * @return The counter value or null if not found
     * @throws DynamoDbException if an error occurs during the operation
     */
    public static Integer getCounterValue(
        DynamoDbClient dynamoDbClient, String tableName, Map<String,
AttributeValue> key, String counterName) {
```

```
        // Define the get parameters
        GetItemRequest request = GetItemRequest.builder()
            .tableName(tableName)
            .key(key)
            .projectionExpression(counterName)
            .build();

        // Perform the get operation
        GetItemResponse response = dynamoDbClient.getItem(request);

        // Return the counter value if it exists, otherwise null
        if (response.item() != null && response.item().containsKey(counterName))
{
            return Integer.parseInt(response.item().get(counterName).n());
        }

        return null;
    }
```

Example usage of atomic counter operations with AWS SDK for Java 2.x.

```
    public static void exampleUsage(DynamoDbClient dynamoDbClient, String
tableName) {
        // Example key
        Map<String, AttributeValue> key = new HashMap<>();
        key.put("ProductId", AttributeValue.builder().s("P12345").build());

        System.out.println("Demonstrating atomic counter operations in
DynamoDB");

        try {
            // Example 1: Increment a counter using ADD
            System.out.println("\nExample 1: Incrementing a counter using ADD");
            UpdateItemResponse addResponse =
incrementCounterWithAdd(dynamoDbClient, tableName, key, "ViewCount", 1);

            System.out.println("Updated counter: " + addResponse.attributes());

            // Example 2: Increment a counter using SET
            System.out.println("\nExample 2: Incrementing a counter using SET");
```

```
            UpdateItemResponse setResponse =
incrementCounterWithSet(dynamoDbClient, tableName, key, "LikeCount", 1);

            System.out.println("Updated counter: " + setResponse.attributes());

            // Example 3: Increment a counter safely
            System.out.println("\nExample 3: Incrementing a counter safely");
            UpdateItemResponse safeResponse =
incrementCounterSafely(dynamoDbClient, tableName, key, "ShareCount", 1);

            System.out.println("Updated counter: " + safeResponse.attributes());

            // Example 4: Decrement a counter safely
            System.out.println("\nExample 4: Decrementing a counter safely");
            try {
                UpdateItemResponse decrementResponse =
                    decrementCounterSafely(dynamoDbClient, tableName, key,
"InventoryCount", 1);

                System.out.println("Updated counter: " +
decrementResponse.attributes());
            } catch (DynamoDbException e) {
                if (e.getMessage().contains("ConditionalCheckFailed")) {
                    System.out.println("Cannot decrement counter below zero");
                } else {
                    throw e;
                }
            }

            // Example 5: Compare ADD vs SET
            System.out.println("\nExample 5: Comparing ADD vs SET");
            Map<String, Object> comparison = compareAddVsSet(dynamoDbClient,
tableName, key);

            if ((boolean) comparison.get("success")) {
                System.out.println("ADD duration: " +
comparison.get("addDuration") + " ns");
                System.out.println("SET duration: " +
comparison.get("setDuration") + " ns");
                System.out.println("ADD response: " +
comparison.get("addResponse"));
                System.out.println("SET response: " +
comparison.get("setResponse"));
            } else {
```

```
                System.out.println("Comparison failed: " +
comparison.get("error"));
            }

            // Explain atomic counter operations
            System.out.println("\nKey points about DynamoDB atomic counter
operations:");
            System.out.println("1. Both ADD and SET can be used for atomic
counters");
            System.out.println("2. ADD is more concise for simple increments");
            System.out.println("3. SET with an expression is more flexible for
complex operations");
            System.out.println("4. Use if_not_exists to handle the case where the
counter doesn't exist yet");
            System.out.println("5. Use condition expressions to prevent counters
from going below zero");
            System.out.println("6. Atomic operations are guaranteed to be
isolated from other writes");
            System.out.println("7. ADD can only be used with number and set data
types");

        } catch (DynamoDbException e) {
            System.err.println("Error: " + e.getMessage());
            e.printStackTrace();
        }
    }
```

- For API details, see [UpdateItem](#) in *AWS SDK for Java 2.x API Reference*.

JavaScript

SDK for JavaScript (v3)

Demonstrate atomic counter operations using AWS SDK for JavaScript.

```
const { DynamoDBClient } = require("@aws-sdk/client-dynamodb");
const {
  DynamoDBDocumentClient,
  UpdateCommand,
  GetCommand
} = require("@aws-sdk/lib-dynamodb");
```

```
/**
 * Increment a counter using the ADD operation.
 *
 * This function demonstrates using the ADD operation for atomic increments.
 * The ADD operation is atomic and is the recommended way to increment counters.
 *
 * @param {Object} config - AWS configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {Object} key - The key of the item to update
 * @param {string} counterName - The name of the counter attribute
 * @param {number} incrementValue - The value to increment by
 * @returns {Promise<Object>} - The response from DynamoDB
 */
async function incrementCounterWithAdd(
  config,
  tableName,
  key,
  counterName,
  incrementValue
) {
  // Initialize the DynamoDB client
  const client = new DynamoDBClient(config);
  const docClient = DynamoDBDocumentClient.from(client);

  // Define the update parameters using ADD
  const params = {
    TableName: tableName,
    Key: key,
    UpdateExpression: `ADD ${counterName} :increment`,
    ExpressionAttributeValues: {
      ":increment": incrementValue
    },
    ReturnValues: "UPDATED_NEW"
  };

  // Perform the update operation
  const response = await docClient.send(new UpdateCommand(params));

  return response;
}

/**
 * Increment a counter using the SET operation with an expression.
 *
```

```
 * This function demonstrates using the SET operation with an expression for
 increments.
 * While this approach works, it's less idiomatic for simple increments than
 using ADD.
 *
 * @param {Object} config - AWS configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {Object} key - The key of the item to update
 * @param {string} counterName - The name of the counter attribute
 * @param {number} incrementValue - The value to increment by
 * @returns {Promise<Object>} - The response from DynamoDB
 */
async function incrementCounterWithSet(
  config,
  tableName,
  key,
  counterName,
  incrementValue
) {
  // Initialize the DynamoDB client
  const client = new DynamoDBClient(config);
  const docClient = DynamoDBDocumentClient.from(client);

  // Define the update parameters using SET with an expression
  const params = {
    TableName: tableName,
    Key: key,
    UpdateExpression: `SET ${counterName} = ${counterName} + :increment`,
    ExpressionAttributeValues: {
      ":increment": incrementValue
    },
    ReturnValues: "UPDATED_NEW"
  };

  // Perform the update operation
  const response = await docClient.send(new UpdateCommand(params));

  return response;
}

/**
 * Increment a counter safely, handling the case where the counter might not
 exist.
 *
```

```
 * This function demonstrates using the if_not_exists function with SET to safely
 * increment a counter that might not exist yet.
 *
 * @param {Object} config - AWS configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {Object} key - The key of the item to update
 * @param {string} counterName - The name of the counter attribute
 * @param {number} incrementValue - The value to increment by
 * @param {number} defaultValue - The default value if the counter doesn't exist
 * @returns {Promise<Object>} - The response from DynamoDB
 */
async function incrementCounterSafely(
  config,
  tableName,
  key,
  counterName,
  incrementValue,
  defaultValue = 0
) {
  // Initialize the DynamoDB client
  const client = new DynamoDBClient(config);
  const docClient = DynamoDBDocumentClient.from(client);

  // Define the update parameters using SET with if_not_exists
  const params = {
    TableName: tableName,
    Key: key,
    UpdateExpression: `SET ${counterName} =
 if_not_exists(${counterName}, :default) + :increment`,
    ExpressionAttributeValues: {
      ":increment": incrementValue,
      ":default": defaultValue
    },
    ReturnValues: "UPDATED_NEW"
  };

  // Perform the update operation
  const response = await docClient.send(new UpdateCommand(params));

  return response;
}

/**
 * Increment a counter with optimistic locking to prevent race conditions.
```

```
 *
 * This function demonstrates using a condition expression to implement
 optimistic
 * locking, which prevents race conditions when multiple processes try to update
 * the same counter.
 *
 * @param {Object} config - AWS configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {Object} key - The key of the item to update
 * @param {string} counterName - The name of the counter attribute
 * @param {number} incrementValue - The value to increment by
 * @param {number} expectedValue - The expected current value of the counter
 * @returns {Promise<Object>} - The response from DynamoDB
 */
async function incrementCounterWithLocking(
  config,
  tableName,
  key,
  counterName,
  incrementValue,
  expectedValue
) {
  // Initialize the DynamoDB client
  const client = new DynamoDBClient(config);
  const docClient = DynamoDBDocumentClient.from(client);

  // Define the update parameters with a condition expression
  const params = {
    TableName: tableName,
    Key: key,
    UpdateExpression: `SET ${counterName} = ${counterName} + :increment`,
    ConditionExpression: `${counterName} = :expected`,
    ExpressionAttributeValues: {
      ":increment": incrementValue,
      ":expected": expectedValue
    },
    ReturnValues: "UPDATED_NEW"
  };

  try {
    // Perform the update operation
    const response = await docClient.send(new UpdateCommand(params));
    return {
      success: true,
```

```
      data: response
    };
  } catch (error) {
    // Check if the error is due to the condition check failing
    if (error.name === "ConditionalCheckFailedException") {
      return {
        success: false,
        error: "Optimistic locking failed: the counter value has changed"
      };
    }
    // Re-throw other errors
    throw error;
  }
}

/**
 * Get the current value of a counter.
 *
 * Helper function to retrieve the current value of a counter attribute.
 *
 * @param {Object} config - AWS configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {Object} key - The key of the item to get
 * @param {string} counterName - The name of the counter attribute
 * @returns {Promise<number|null>} - The current counter value or null if not
 found
 */
async function getCounterValue(
  config,
  tableName,
  key,
  counterName
) {
  // Initialize the DynamoDB client
  const client = new DynamoDBClient(config);
  const docClient = DynamoDBDocumentClient.from(client);

  // Define the get parameters
  const params = {
    TableName: tableName,
    Key: key
  };

  // Perform the get operation
```

```
  const response = await docClient.send(new GetCommand(params));

  // Return the counter value if it exists, otherwise null
  return response.Item && counterName in response.Item
    ? response.Item[counterName]
    : null;
}
```

Example usage of atomic counter operations with AWS SDK for JavaScript.

```
/**
 * Example of how to use the atomic counter operations.
 */
async function exampleUsage() {
  // Example parameters
  const config = { region: "us-west-2" };
  const tableName = "Products";
  const key = { ProductId: "P12345" };
  const counterName = "ViewCount";
  const incrementValue = 1;

  console.log("Demonstrating different approaches to increment counters in
 DynamoDB");

  try {
    // Example 1: Using ADD operation (recommended for simple increments)
    console.log("\nExample 1: Incrementing counter with ADD operation");
    const response1 = await incrementCounterWithAdd(
      config,
      tableName,
      key,
      counterName,
      incrementValue
    );

    console.log(`Counter incremented to: ${response1.Attributes[counterName]}`);

    // Example 2: Using SET operation with an expression
    console.log("\nExample 2: Incrementing counter with SET operation");
    const response2 = await incrementCounterWithSet(
      config,
      tableName,
```

```
      key,
      counterName,
      incrementValue
    );

    console.log(`Counter incremented to: ${response2.Attributes[counterName]}`);

    // Example 3: Safely incrementing a counter that might not exist
    console.log("\nExample 3: Safely incrementing counter that might not exist");
    const newKey = { ProductId: "P67890" };
    const response3 = await incrementCounterSafely(
      config,
      tableName,
      newKey,
      counterName,
      incrementValue,
      0
    );

    console.log(`Counter initialized and incremented to:
${response3.Attributes[counterName]}`);

    // Example 4: Incrementing with optimistic locking
    console.log("\nExample 4: Incrementing with optimistic locking");

    // First, get the current counter value
    const currentValue = await getCounterValue(config, tableName, key,
counterName);
    console.log(`Current counter value: ${currentValue}`);

    // Then, try to increment with optimistic locking
    const response4 = await incrementCounterWithLocking(
      config,
      tableName,
      key,
      counterName,
      incrementValue,
      currentValue
    );

    if (response4.success) {
      console.log(`Counter successfully incremented to:
${response4.data.Attributes[counterName]}`);
    } else {
```

```
      console.log(response4.error);
    }

    // Explain the differences between ADD and SET
    console.log("\nKey differences between ADD and SET for counter operations:");
    console.log("1. ADD is more concise and idiomatic for simple increments");
    console.log("2. SET with expressions is more flexible for complex
 operations");
    console.log("3. Both operations are atomic and safe for concurrent updates");
    console.log("4. SET with if_not_exists is required when the attribute might
 not exist");
    console.log("5. Optimistic locking can be added to either approach for
 additional safety");

  } catch (error) {
    console.error("Error:", error);
  }
}
```

- For API details, see UpdateItem in *AWS SDK for JavaScript API Reference*.

Python

SDK for Python (Boto3)

Demonstrate atomic counter operations using AWS SDK for Python (Boto3).

```
import boto3
from botocore.exceptions import ClientError
from typing import Any, Dict, Union


def increment_counter_with_add(
    table_name: str, key: Dict[str, Any], counter_name: str, increment_value: int
 = 1
) -> Dict[str, Any]:
    """
    Increment a counter attribute using the ADD operation.

    This function demonstrates the atomic ADD operation, which is ideal for
    incrementing counters without the risk of race conditions.
```

```
    Args:
        table_name (str): The name of the DynamoDB table.
        key (Dict[str, Any]): The primary key of the item to update.
        counter_name (str): The name of the counter attribute.
        increment_value (int, optional): The value to increment by. Defaults to
 1.

    Returns:
        Dict[str, Any]: The response from DynamoDB containing the updated
 attribute values.
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # Use the ADD operation to atomically increment the counter
    response = table.update_item(
        Key=key,
        UpdateExpression="ADD #counter :increment",
        ExpressionAttributeNames={"#counter": counter_name},
        ExpressionAttributeValues={":increment": increment_value},
        ReturnValues="UPDATED_NEW",
    )

    return response


def increment_counter_with_set(
    table_name: str, key: Dict[str, Any], counter_name: str, increment_value: int
 = 1
) -> Dict[str, Any]:
    """
    Increment a counter attribute using the SET operation with an expression.

    This function demonstrates using SET with an expression to increment a
 counter.
    While this works, it's generally recommended to use ADD for simple
 increments.

    Args:
        table_name (str): The name of the DynamoDB table.
        key (Dict[str, Any]): The primary key of the item to update.
        counter_name (str): The name of the counter attribute.
```

```
        increment_value (int, optional): The value to increment by. Defaults to
 1.

    Returns:
        Dict[str, Any]: The response from DynamoDB containing the updated
 attribute values.
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # Use the SET operation with an expression to increment the counter
    response = table.update_item(
        Key=key,
        UpdateExpression="SET #counter = #counter + :increment",
        ExpressionAttributeNames={"#counter": counter_name},
        ExpressionAttributeValues={":increment": increment_value},
        ReturnValues="UPDATED_NEW",
    )

    return response


def increment_counter_safely(
    table_name: str,
    key: Dict[str, Any],
    counter_name: str,
    increment_value: int = 1,
    initial_value: int = 0,
) -> Dict[str, Any]:
    """
    Increment a counter attribute safely, handling the case where it might not
 exist.

    This function demonstrates a best practice for incrementing counters by using
    the if_not_exists function to handle the case where the counter doesn't exist
 yet.

    Args:
        table_name (str): The name of the DynamoDB table.
        key (Dict[str, Any]): The primary key of the item to update.
        counter_name (str): The name of the counter attribute.
        increment_value (int, optional): The value to increment by. Defaults to
 1.
```

```
        initial_value (int, optional): The initial value if the counter doesn't
 exist. Defaults to 0.

    Returns:
        Dict[str, Any]: The response from DynamoDB containing the updated
 attribute values.
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # Use SET with if_not_exists to safely increment the counter
    response = table.update_item(
        Key=key,
        UpdateExpression="SET #counter = if_not_exists(#counter, :initial)
 + :increment",
        ExpressionAttributeNames={"#counter": counter_name},
        ExpressionAttributeValues={":increment": increment_value, ":initial":
 initial_value},
        ReturnValues="UPDATED_NEW",
    )

    return response


def atomic_conditional_increment(
    table_name: str,
    key: Dict[str, Any],
    counter_name: str,
    condition_attribute: str,
    condition_value: Any,
    increment_value: int = 1,
) -> Union[Dict[str, Any], None]:
    """
    Atomically increment a counter only if a condition is met.

    This function demonstrates combining atomic counter operations with
    conditional expressions for more complex update scenarios.

    Args:
        table_name (str): The name of the DynamoDB table.
        key (Dict[str, Any]): The primary key of the item to update.
        counter_name (str): The name of the counter attribute.
        condition_attribute (str): The attribute to check in the condition.
```

```
        condition_value (Any): The value to compare against.
        increment_value (int, optional): The value to increment by. Defaults to
 1.

    Returns:
        Optional[Dict[str, Any]]: The response from DynamoDB if successful, None
 if condition failed.
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    try:
        # Use ADD with a condition expression
        response = table.update_item(
            Key=key,
            UpdateExpression="ADD #counter :increment",
            ConditionExpression="#condition = :value",
            ExpressionAttributeNames={"#counter": counter_name, "#condition":
 condition_attribute},
            ExpressionAttributeValues={":increment": increment_value, ":value":
 condition_value},
            ReturnValues="UPDATED_NEW",
        )
        return response
    except ClientError as e:
        if e.response["Error"]["Code"] == "ConditionalCheckFailedException":
            # Condition was not met
            return None
        else:
            # Other error occurred
            raise
```

Example usage of atomic counter operations with AWS SDK for Python (Boto3).

```
def example_usage():
    """Example of how to use the atomic counter operations functions."""
    # Example parameters
    table_name = "GameScores"
    key = {"UserId": "user123", "GameId": "game456"}
```

```
    counter_name = "Score"

    print("Example 1: Incrementing a counter with ADD operation")
    try:
        response = increment_counter_with_add(
            table_name=table_name, key=key, counter_name=counter_name,
increment_value=10
        )
        print(
            f"Counter incremented successfully. New value:
{response.get('Attributes', {}).get(counter_name)}"
        )
    except Exception as e:
        print(f"Error incrementing counter with ADD: {e}")

    print("\nExample 2: Incrementing a counter with SET operation")
    try:
        response = increment_counter_with_set(
            table_name=table_name, key=key, counter_name=counter_name,
increment_value=5
        )
        print(
            f"Counter incremented successfully. New value:
{response.get('Attributes', {}).get(counter_name)}"
        )
    except Exception as e:
        print(f"Error incrementing counter with SET: {e}")

    print("\nExample 3: Safely incrementing a counter that might not exist")
    try:
        new_key = {"UserId": "newuser789", "GameId": "game456"}
        response = increment_counter_safely(
            table_name=table_name,
            key=new_key,
            counter_name=counter_name,
            increment_value=15,
            initial_value=100,
        )
        print(
            f"Counter safely incremented. New value: {response.get('Attributes',
{}).get(counter_name)}"
        )
    except Exception as e:
        print(f"Error safely incrementing counter: {e}")
```

```
    print("\nExample 4: Conditional counter increment")
    try:
        # Fix for mypy: Handle the case where response might be None
        result = atomic_conditional_increment(
            table_name=table_name,
            key=key,
            counter_name="Achievements",
            condition_attribute="Level",
            condition_value=5,
            increment_value=1,
        )

        if result is not None:
            print(
                f"Conditional increment succeeded. New value:
{result.get('Attributes', {}).get('Achievements')}"
            )
        else:
            print("Conditional increment failed because condition was not met.")
        if response:
            print(
                f"Conditional increment succeeded. New value:
{response.get('Attributes', {}).get('Achievements')}"
            )
        else:
            print("Conditional increment failed because condition was not met.")
    except Exception as e:
        print(f"Error with conditional increment: {e}")

    print("\nComparison of ADD vs SET for counter operations:")
    print("1. ADD is specifically designed for atomic numeric increments and set
operations")
    print("2. SET with an expression can be used for more complex calculations")
    print("3. Both operations are atomic, preventing race conditions")
    print("4. ADD is more concise for simple increments")
    print("5. SET with if_not_exists() is recommended when the attribute might
not exist")
    print("6. For counters, ADD is generally preferred for clarity and
simplicity")
```

- For API details, see UpdateItem in *AWS SDK for Python (Boto3) API Reference*.

For a complete list of AWS SDK developer guides and code examples, see Using DynamoDB with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use conditional operations in DynamoDB with an AWS SDK

The following code examples show how to use conditional operations in DynamoDB.

- Implement conditional writes to prevent overwriting data.
- Use condition expressions to enforce business rules.
- Handle conditional check failures gracefully.

Java

SDK for Java 2.x

Demonstrate conditional operations using AWS SDK for Java 2.x.

```
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import
 software.amazon.awssdk.services.dynamodb.model.ConditionalCheckFailedException;
import software.amazon.awssdk.services.dynamodb.model.DeleteItemRequest;
import software.amazon.awssdk.services.dynamodb.model.DeleteItemResponse;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.GetItemRequest;
import software.amazon.awssdk.services.dynamodb.model.GetItemResponse;
import software.amazon.awssdk.services.dynamodb.model.ReturnValue;
import software.amazon.awssdk.services.dynamodb.model.UpdateItemRequest;
import software.amazon.awssdk.services.dynamodb.model.UpdateItemResponse;

import java.util.HashMap;
import java.util.Map;

    /**
     * Performs a conditional update on an item.
     *
     * <p>This method demonstrates how to use a condition expression to update an
 item
```

```
     * only if a specific condition is met.
     *
     * @param dynamoDbClient The DynamoDB client
     * @param tableName The name of the DynamoDB table
     * @param key The key of the item to update
     * @param conditionAttribute The attribute to check in the condition
     * @param conditionValue The value to compare against
     * @param updateAttribute The attribute to update
     * @param updateValue The new value to set
     * @return Map containing the operation result and status
     */
    public static Map<String, Object> conditionalUpdate(
        DynamoDbClient dynamoDbClient,
        String tableName,
        Map<String, AttributeValue> key,
        String conditionAttribute,
        AttributeValue conditionValue,
        String updateAttribute,
        AttributeValue updateValue) {

        Map<String, Object> result = new HashMap<>();

        try {
            // Define the update parameters
            UpdateItemRequest request = UpdateItemRequest.builder()
                .tableName(tableName)
                .key(key)
                .updateExpression("SET #updateAttr = :updateVal")
                .conditionExpression("#condAttr = :condVal")
                .expressionAttributeNames(Map.of(
                    "#condAttr", conditionAttribute,
                    "#updateAttr", updateAttribute))
                .expressionAttributeValues(Map.of(
                    ":condVal", conditionValue,
                    ":updateVal", updateValue))
                .returnValues(ReturnValue.UPDATED_NEW)
                .build();

            // Perform the update operation
            UpdateItemResponse response = dynamoDbClient.updateItem(request);

            // Record success result
            result.put("success", true);
            result.put("message", "Condition was met and update was performed");
```

```
            result.put("attributes", response.attributes());

        } catch (ConditionalCheckFailedException e) {
            // Record failure due to condition not being met
            result.put("success", false);
            result.put("message", "Condition was not met, update was not
performed");
            result.put("error", "ConditionalCheckFailedException");

        } catch (DynamoDbException e) {
            // Record failure due to other errors
            result.put("success", false);
            result.put("message", "Error occurred: " + e.getMessage());
            result.put("error", e.getClass().getSimpleName());
        }

        return result;
    }

    /**
     * Performs a conditional delete on an item.
     *
     * <p>This method demonstrates how to use a condition expression to delete an
item
     * only if a specific condition is met.
     *
     * @param dynamoDbClient The DynamoDB client
     * @param tableName The name of the DynamoDB table
     * @param key The key of the item to delete
     * @param conditionAttribute The attribute to check in the condition
     * @param conditionValue The value to compare against
     * @return Map containing the operation result and status
     */
    public static Map<String, Object> conditionalDelete(
        DynamoDbClient dynamoDbClient,
        String tableName,
        Map<String, AttributeValue> key,
        String conditionAttribute,
        AttributeValue conditionValue) {

        Map<String, Object> result = new HashMap<>();

        try {
            // Define the delete parameters
```

```
            DeleteItemRequest request = DeleteItemRequest.builder()
                .tableName(tableName)
                .key(key)
                .conditionExpression("#condAttr = :condVal")
                .expressionAttributeNames(Map.of("#condAttr",
conditionAttribute))
                .expressionAttributeValues(Map.of(":condVal", conditionValue))
                .returnValues(ReturnValue.ALL_OLD)
                .build();

            // Perform the delete operation
            DeleteItemResponse response = dynamoDbClient.deleteItem(request);

            // Record success result
            result.put("success", true);
            result.put("message", "Condition was met and delete was performed");
            result.put("attributes", response.attributes());

        } catch (ConditionalCheckFailedException e) {
            // Record failure due to condition not being met
            result.put("success", false);
            result.put("message", "Condition was not met, delete was not
performed");
            result.put("error", "ConditionalCheckFailedException");

        } catch (DynamoDbException e) {
            // Record failure due to other errors
            result.put("success", false);
            result.put("message", "Error occurred: " + e.getMessage());
            result.put("error", e.getClass().getSimpleName());
        }

        return result;
    }

    /**
     * Demonstrates optimistic locking using a version attribute.
     *
     * <p>This method shows how to implement optimistic locking by using a
version
     * attribute that is incremented with each update.
     *
     * @param dynamoDbClient The DynamoDB client
     * @param tableName The name of the DynamoDB table
```

```
     * @param key The key of the item to update
     * @param versionAttribute The name of the version attribute
     * @return Map containing the operation result
     */
    public static Map<String, Object> optimisticLockingExample(
        DynamoDbClient dynamoDbClient, String tableName, Map<String,
 AttributeValue> key, String versionAttribute) {

        Map<String, Object> result = new HashMap<>();

        try {
            // Get the current version of the item
            GetItemRequest getRequest = GetItemRequest.builder()
                .tableName(tableName)
                .key(key)
                .projectionExpression(versionAttribute)
                .build();

            GetItemResponse getResponse = dynamoDbClient.getItem(getRequest);

            // Check if the item exists
            if (getResponse.item() == null || !
getResponse.item().containsKey(versionAttribute)) {
                // Item doesn't exist or doesn't have a version attribute
                // Initialize with version 1
                UpdateItemRequest initRequest = UpdateItemRequest.builder()
                    .tableName(tableName)
                    .key(key)
                    .updateExpression("SET #verAttr = :newVer, #dataAttr
 = :data")
                    .expressionAttributeNames(Map.of("#verAttr",
 versionAttribute, "#dataAttr", "Data"))
                    .expressionAttributeValues(Map.of(
                        ":newVer", AttributeValue.builder().n("1").build(),
                        ":data", AttributeValue.builder().s("Initial
 data").build()))
                    .returnValues(ReturnValue.UPDATED_NEW)
                    .build();

                UpdateItemResponse initResponse =
 dynamoDbClient.updateItem(initRequest);

                result.put("operation", "initialize");
                result.put("success", true);
```

```
                result.put("attributes", initResponse.attributes());

                return result;
            }

            // Get the current version number
            int currentVersion =
                Integer.parseInt(getResponse.item().get(versionAttribute).n());
            int newVersion = currentVersion + 1;

            // Update the item with a condition on the version
            UpdateItemRequest updateRequest = UpdateItemRequest.builder()
                .tableName(tableName)
                .key(key)
                .updateExpression("SET #verAttr = :newVer, #dataAttr = :newData")
                .conditionExpression("#verAttr = :curVer")
                .expressionAttributeNames(Map.of("#verAttr", versionAttribute,
"#dataAttr", "Data"))
                .expressionAttributeValues(Map.of(
                    ":curVer",
                        AttributeValue.builder()
                            .n(String.valueOf(currentVersion))
                            .build(),
                    ":newVer",

AttributeValue.builder().n(String.valueOf(newVersion)).build(),
                    ":newData",
                        AttributeValue.builder()
                            .s("Updated data at version " + newVersion)
                            .build()))
                .returnValues(ReturnValue.UPDATED_NEW)
                .build();

            UpdateItemResponse updateResponse =
dynamoDbClient.updateItem(updateRequest);

            // Record success result
            result.put("operation", "update");
            result.put("success", true);
            result.put("oldVersion", currentVersion);
            result.put("newVersion", newVersion);
            result.put("attributes", updateResponse.attributes());

        } catch (ConditionalCheckFailedException e) {
```

```
            // Record failure due to version mismatch
            result.put("operation", "update");
            result.put("success", false);
            result.put("message", "Version mismatch, another process may have
updated the item");
            result.put("error", "ConditionalCheckFailedException");

        } catch (DynamoDbException e) {
            // Record failure due to other errors
            result.put("operation", "update");
            result.put("success", false);
            result.put("message", "Error occurred: " + e.getMessage());
            result.put("error", e.getClass().getSimpleName());
        }

        return result;
    }

    /**
     * Performs a conditional update with multiple conditions.
     *
     * <p>This method demonstrates how to use multiple conditions in a condition
expression.
     *
     * @param dynamoDbClient The DynamoDB client
     * @param tableName The name of the DynamoDB table
     * @param key The key of the item to update
     * @param conditions Map of attribute names to values for conditions
     * @param updateAttribute The attribute to update
     * @param updateValue The new value to set
     * @return Map containing the operation result and status
     */
    public static Map<String, Object> conditionalUpdateWithMultipleConditions(
        DynamoDbClient dynamoDbClient,
        String tableName,
        Map<String, AttributeValue> key,
        Map<String, AttributeValue> conditions,
        String updateAttribute,
        AttributeValue updateValue) {

        Map<String, Object> result = new HashMap<>();

        try {
            // Build the condition expression and attribute names/values
```

```
            StringBuilder conditionExpression = new StringBuilder();
            Map<String, String> expressionAttributeNames = new HashMap<>();
            Map<String, AttributeValue> expressionAttributeValues = new
HashMap<>();

            // Add update attribute
            expressionAttributeNames.put("#updateAttr", updateAttribute);
            expressionAttributeValues.put(":updateVal", updateValue);

            // Add conditions
            int i = 0;
            for (Map.Entry<String, AttributeValue> condition :
conditions.entrySet()) {
                String attrName = condition.getKey();
                AttributeValue attrValue = condition.getValue();

                String nameKey = "#cond" + i;
                String valueKey = ":val" + i;

                expressionAttributeNames.put(nameKey, attrName);
                expressionAttributeValues.put(valueKey, attrValue);

                // Add AND between conditions (except for the first one)
                if (i > 0) {
                    conditionExpression.append(" AND ");
                }

                conditionExpression.append(nameKey).append(" =
").append(valueKey);
                i++;
            }

            // Define the update parameters
            UpdateItemRequest request = UpdateItemRequest.builder()
                .tableName(tableName)
                .key(key)
                .updateExpression("SET #updateAttr = :updateVal")
                .conditionExpression(conditionExpression.toString())
                .expressionAttributeNames(expressionAttributeNames)
                .expressionAttributeValues(expressionAttributeValues)
                .returnValues(ReturnValue.UPDATED_NEW)
                .build();

            // Perform the update operation
```

```
            UpdateItemResponse response = dynamoDbClient.updateItem(request);

            // Record success result
            result.put("success", true);
            result.put("message", "All conditions were met and update was
performed");
            result.put("attributes", response.attributes());

        } catch (ConditionalCheckFailedException e) {
            // Record failure due to condition not being met
            result.put("success", false);
            result.put("message", "One or more conditions were not met, update
was not performed");
            result.put("error", "ConditionalCheckFailedException");

        } catch (DynamoDbException e) {
            // Record failure due to other errors
            result.put("success", false);
            result.put("message", "Error occurred: " + e.getMessage());
            result.put("error", e.getClass().getSimpleName());
        }

        return result;
    }
```

Example usage of conditional operations with AWS SDK for Java 2.x.

```
    public static void exampleUsage(DynamoDbClient dynamoDbClient, String
tableName) {
        // Example key
        Map<String, AttributeValue> key = new HashMap<>();
        key.put("ProductId", AttributeValue.builder().s("P12345").build());

        System.out.println("Demonstrating conditional operations in DynamoDB");

        try {
            // Example 1: Conditional update
            System.out.println("\nExample 1: Conditional update");
            Map<String, Object> updateResult = conditionalUpdate(
                dynamoDbClient,
                tableName,
                key,
```

```
                "InStock",
                AttributeValue.builder().bool(true).build(),
                "Status",
                AttributeValue.builder().s("Available").build());

            System.out.println("Update result: " + updateResult.get("message"));
            if ((boolean) updateResult.get("success")) {
                System.out.println("Updated attributes: " +
updateResult.get("attributes"));
            }

            // Example 2: Conditional delete
            System.out.println("\nExample 2: Conditional delete");
            Map<String, Object> deleteResult = conditionalDelete(
                dynamoDbClient,
                tableName,
                key,
                "Status",
                AttributeValue.builder().s("Discontinued").build());

            System.out.println("Delete result: " + deleteResult.get("message"));
            if ((boolean) deleteResult.get("success")) {
                System.out.println("Deleted item: " +
deleteResult.get("attributes"));
            }

            // Example 3: Optimistic locking
            System.out.println("\nExample 3: Optimistic locking");
            Map<String, Object> lockingResult =
optimisticLockingExample(dynamoDbClient, tableName, key, "Version");

            System.out.println("Optimistic locking result:");
            System.out.println("  Operation: " + lockingResult.get("operation"));
            System.out.println("  Success: " + lockingResult.get("success"));
            if (lockingResult.get("operation").equals("update") && (boolean)
lockingResult.get("success")) {
                System.out.println("  Old version: " +
lockingResult.get("oldVersion"));
                System.out.println("  New version: " +
lockingResult.get("newVersion"));
            }
            System.out.println("  Attributes: " +
lockingResult.get("attributes"));
```

```
            // Example 4: Multiple conditions
            System.out.println("\nExample 4: Multiple conditions");
            Map<String, AttributeValue> conditions = new HashMap<>();
            conditions.put("Price",
AttributeValue.builder().n("199.99").build());
            conditions.put("Category",
AttributeValue.builder().s("Electronics").build());

            Map<String, Object> multiConditionResult =
conditionalUpdateWithMultipleConditions(
                dynamoDbClient,
                tableName,
                key,
                conditions,
                "OnSale",
                AttributeValue.builder().bool(true).build());

            System.out.println("Multiple conditions result: " +
multiConditionResult.get("message"));
            if ((boolean) multiConditionResult.get("success")) {
                System.out.println("Updated attributes: " +
multiConditionResult.get("attributes"));
            }

            // Explain conditional operations
            System.out.println("\nKey points about DynamoDB conditional
operations:");
            System.out.println("1. Conditional operations only succeed if the
condition is met");
            System.out.println("2. ConditionalCheckFailedException is thrown when
the condition fails");
            System.out.println("3. No changes are made to the item if the
condition fails");
            System.out.println("4. Conditions can be used with update, delete,
and put operations");
            System.out.println("5. Multiple conditions can be combined with AND
and OR");
            System.out.println("6. Optimistic locking can be implemented using a
version attribute");
            System.out.println(
                "7. Conditional operations consume the same amount of write
capacity whether they succeed or fail");

        } catch (DynamoDbException e) {
```

```
            System.err.println("Error: " + e.getMessage());
            e.printStackTrace();
        }
    }
```

- For API details, see the following topics in *AWS SDK for Java 2.x API Reference*.
  - DeleteItem
  - PutItem
  - UpdateItem

JavaScript

SDK for JavaScript (v3)

Demonstrate conditional operations using AWS SDK for JavaScript.

```
const { DynamoDBClient } = require("@aws-sdk/client-dynamodb");
const {
  DynamoDBDocumentClient,
  UpdateCommand,
  DeleteCommand,
  GetCommand,
  PutCommand
} = require("@aws-sdk/lib-dynamodb");

/**
 * Perform a conditional update operation.
 *
 * This function demonstrates how to update an item only if a condition is met.
 *
 * @param {Object} config - AWS configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {Object} key - The key of the item to update
 * @param {string} conditionAttribute - The attribute to check in the condition
 * @param {any} conditionValue - The value to compare against
 * @param {string} updateAttribute - The attribute to update
 * @param {any} updateValue - The new value to set
 * @returns {Promise<Object>} - Result of the operation
 */
async function conditionalUpdate(
  config,
```

```
  tableName,
  key,
  conditionAttribute,
  conditionValue,
  updateAttribute,
  updateValue
) {
  // Initialize the DynamoDB client
  const client = new DynamoDBClient(config);
  const docClient = DynamoDBDocumentClient.from(client);

  // Define the update parameters with a condition expression
  const params = {
    TableName: tableName,
    Key: key,
    UpdateExpression: `SET ${updateAttribute} = :value`,
    ConditionExpression: `${conditionAttribute} = :condition`,
    ExpressionAttributeValues: {
      ":value": updateValue,
      ":condition": conditionValue
    },
    ReturnValues: "UPDATED_NEW"
  };

  try {
    // Perform the update operation
    const response = await docClient.send(new UpdateCommand(params));

    return {
      success: true,
      message: "Condition was met and update was performed",
      updatedAttributes: response.Attributes
    };
  } catch (error) {
    // Check if the error is due to the condition check failing
    if (error.name === "ConditionalCheckFailedException") {
      return {
        success: false,
        message: "Condition was not met, update was not performed",
        error: "ConditionalCheckFailedException"
      };
    }

    // Re-throw other errors
```

```
    throw error;
  }
}

/**
 * Perform a conditional delete operation.
 *
 * This function demonstrates how to delete an item only if a condition is met.
 *
 * @param {Object} config - AWS configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {Object} key - The key of the item to delete
 * @param {string} conditionAttribute - The attribute to check in the condition
 * @param {any} conditionValue - The value to compare against
 * @returns {Promise<Object>} - Result of the operation
 */
async function conditionalDelete(
  config,
  tableName,
  key,
  conditionAttribute,
  conditionValue
) {
  // Initialize the DynamoDB client
  const client = new DynamoDBClient(config);
  const docClient = DynamoDBDocumentClient.from(client);

  // Define the delete parameters with a condition expression
  const params = {
    TableName: tableName,
    Key: key,
    ConditionExpression: `${conditionAttribute} = :condition`,
    ExpressionAttributeValues: {
      ":condition": conditionValue
    },
    ReturnValues: "ALL_OLD"
  };

  try {
    // Perform the delete operation
    const response = await docClient.send(new DeleteCommand(params));

    return {
      success: true,
```

```
      message: "Condition was met and item was deleted",
      deletedItem: response.Attributes
    };
  } catch (error) {
    // Check if the error is due to the condition check failing
    if (error.name === "ConditionalCheckFailedException") {
      return {
        success: false,
        message: "Condition was not met, item was not deleted",
        error: "ConditionalCheckFailedException"
      };
    }

    // Re-throw other errors
    throw error;
  }
}

/**
 * Implement optimistic locking with a version number.
 *
 * This function demonstrates how to use a version number for optimistic locking
 * to prevent race conditions when multiple processes update the same item.
 *
 * @param {Object} config - AWS configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {Object} key - The key of the item to update
 * @param {Object} updates - The attributes to update
 * @param {number} expectedVersion - The expected current version number
 * @returns {Promise<Object>} - Result of the operation
 */
async function updateWithOptimisticLocking(
  config,
  tableName,
  key,
  updates,
  expectedVersion
) {
  // Initialize the DynamoDB client
  const client = new DynamoDBClient(config);
  const docClient = DynamoDBDocumentClient.from(client);

  // Build the update expression
  const updateExpressions = [];
```

```
  const expressionAttributeValues = {
    ":expectedVersion": expectedVersion,
    ":newVersion": expectedVersion + 1
  };

  // Add each update to the expression
  Object.entries(updates).forEach(([attribute, value], index) => {
    updateExpressions.push(`${attribute} = :val${index}`);
    expressionAttributeValues[`:val${index}`] = value;
  });

  // Add the version update
  updateExpressions.push("version = :newVersion");

  // Define the update parameters with a condition expression
  const params = {
    TableName: tableName,
    Key: key,
    UpdateExpression: `SET ${updateExpressions.join(", ")}`,
    ConditionExpression: "version = :expectedVersion",
    ExpressionAttributeValues: expressionAttributeValues,
    ReturnValues: "UPDATED_NEW"
  };

  try {
    // Perform the update operation
    const response = await docClient.send(new UpdateCommand(params));

    return {
      success: true,
      message: "Update succeeded with optimistic locking",
      newVersion: expectedVersion + 1,
      updatedAttributes: response.Attributes
    };
  } catch (error) {
    // Check if the error is due to the condition check failing
    if (error.name === "ConditionalCheckFailedException") {
      return {
        success: false,
        message: "Optimistic locking failed: the item was modified by another
process",
        error: "ConditionalCheckFailedException"
      };
    }
```

```
    // Re-throw other errors
    throw error;
  }
}

/**
 * Implement a conditional write that creates an item only if it doesn't exist.
 *
 * This function demonstrates how to use attribute_not_exists to create an item
 * only if it doesn't already exist (similar to an "INSERT IF NOT EXISTS"
 operation).
 *
 * @param {Object} config - AWS configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {Object} item - The item to create
 * @returns {Promise<Object>} - Result of the operation
 */
async function createIfNotExists(
  config,
  tableName,
  item
) {
  // Initialize the DynamoDB client
  const client = new DynamoDBClient(config);
  const docClient = DynamoDBDocumentClient.from(client);

  // Extract the primary key attributes
  const keyAttributes = Object.keys(item).filter(attr =>
    attr === "id" || attr === "ID" || attr === "Id" ||
    attr.endsWith("Id") || attr.endsWith("ID") ||
    attr.endsWith("Key")
  );

  if (keyAttributes.length === 0) {
    throw new Error("Could not determine primary key attributes");
  }

  // Create a condition expression that checks if the item doesn't exist
  const conditionExpression = `attribute_not_exists(${keyAttributes[0]})`;

  // Define the put parameters with a condition expression
  const params = {
    TableName: tableName,
```

```
    Item: item,
    ConditionExpression: conditionExpression
  };

  try {
    // Perform the put operation
    await docClient.send(new PutCommand(params));

    return {
      success: true,
      message: "Item was created because it didn't exist",
      item
    };
  } catch (error) {
    // Check if the error is due to the condition check failing
    if (error.name === "ConditionalCheckFailedException") {
      return {
        success: false,
        message: "Item already exists, creation was skipped",
        error: "ConditionalCheckFailedException"
      };
    }

    // Re-throw other errors
    throw error;
  }
}

/**
 * Get the current value of an item.
 *
 * Helper function to retrieve the current value of an item.
 *
 * @param {Object} config - AWS configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {Object} key - The key of the item to get
 * @returns {Promise<Object|null>} - The item or null if not found
 */
async function getItem(
  config,
  tableName,
  key
) {
  // Initialize the DynamoDB client
```

```
  const client = new DynamoDBClient(config);
  const docClient = DynamoDBDocumentClient.from(client);

  // Define the get parameters
  const params = {
    TableName: tableName,
    Key: key
  };

  // Perform the get operation
  const response = await docClient.send(new GetCommand(params));

  // Return the item if it exists, otherwise null
  return response.Item || null;
}
```

Example usage of conditional operations with AWS SDK for JavaScript.

```
/**
 * Example of how to use conditional operations.
 */
async function exampleUsage() {
  // Example parameters
  const config = { region: "us-west-2" };
  const tableName = "Products";
  const key = { ProductId: "P12345" };

  console.log("Demonstrating conditional operations in DynamoDB");

  try {
    // Example 1: Conditional update based on attribute value
    console.log("\nExample 1: Conditional update based on attribute value");
    const updateResult = await conditionalUpdate(
      config,
      tableName,
      key,
      "Category",
      "Electronics",
      "Price",
      299.99
    );
```

```
    console.log(`Result: ${updateResult.message}`);
    if (updateResult.success) {
      console.log("Updated attributes:", updateResult.updatedAttributes);
    }

    // Example 2: Conditional delete based on attribute value
    console.log("\nExample 2: Conditional delete based on attribute value");
    const deleteResult = await conditionalDelete(
      config,
      tableName,
      key,
      "InStock",
      false
    );

    console.log(`Result: ${deleteResult.message}`);
    if (deleteResult.success) {
      console.log("Deleted item:", deleteResult.deletedItem);
    }

    // Example 3: Optimistic locking with version number
    console.log("\nExample 3: Optimistic locking with version number");

    // First, get the current item to check its version
    const currentItem = await getItem(config, tableName, { ProductId:
"P67890" });
    const currentVersion = currentItem ? (currentItem.version || 0) : 0;

    console.log(`Current version: ${currentVersion}`);

    // Then, update with optimistic locking
    const lockingResult = await updateWithOptimisticLocking(
      config,
      tableName,
      { ProductId: "P67890" },
      {
        Name: "Updated Product Name",
        Description: "This is an updated description"
      },
      currentVersion
    );

    console.log(`Result: ${lockingResult.message}`);
    if (lockingResult.success) {
```

```
      console.log(`New version: ${lockingResult.newVersion}`);
      console.log("Updated attributes:", lockingResult.updatedAttributes);
    }

    // Example 4: Create item only if it doesn't exist
    console.log("\nExample 4: Create item only if it doesn't exist");
    const createResult = await createIfNotExists(
      config,
      tableName,
      {
        ProductId: "P99999",
        Name: "New Product",
        Category: "Accessories",
        Price: 19.99,
        InStock: true
      }
    );

    console.log(`Result: ${createResult.message}`);
    if (createResult.success) {
      console.log("Created item:", createResult.item);
    }

    // Explain conditional operations
    console.log("\nKey points about conditional operations:");
    console.log("1. Conditional operations only succeed if the condition is
 met");
    console.log("2. ConditionalCheckFailedException indicates the condition
 wasn't met");
    console.log("3. Optimistic locking prevents race conditions in concurrent
 updates");
    console.log("4. attribute_exists and attribute_not_exists are useful for
 checking if attributes are present");
    console.log("5. Conditional operations are atomic - they either succeed
 completely or fail completely");
    console.log("6. You can use any valid comparison operators and functions in
 condition expressions");
    console.log("7. Conditional operations don't consume write capacity if the
 condition fails");

  } catch (error) {
    console.error("Error:", error);
  }
}
```

- For API details, see the following topics in *AWS SDK for JavaScript API Reference*.
  - DeleteItem
  - PutItem
  - UpdateItem

Python

SDK for Python (Boto3)

Demonstrate conditional operations using AWS SDK for Python (Boto3).

```
import boto3
from botocore.exceptions import ClientError
from typing import Any, Dict, Optional, Tuple, Union


def conditional_update(
    table_name: str,
    key: Dict[str, Any],
    condition_attribute: str,
    condition_value: Any,
    update_attribute: str,
    update_value: Any,
) -> Tuple[bool, Optional[Dict[str, Any]]]:
    """
    Update an item only if a condition is met.

    This function demonstrates how to perform a conditional update operation
    and determine if the condition was met.

    Args:
        table_name (str): The name of the DynamoDB table.
        key (Dict[str, Any]): The primary key of the item to update.
        condition_attribute (str): The attribute to check in the condition.
        condition_value (Any): The value to compare against.
        update_attribute (str): The attribute to update.
        update_value (Any): The new value to set.

    Returns:
        Tuple[bool, Optional[Dict[str, Any]]]: A tuple containing:
```

```
            - A boolean indicating if the update succeeded
            - The response from DynamoDB if successful, None otherwise
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    try:
        # Perform the conditional update
        response = table.update_item(
            Key=key,
            UpdateExpression="SET #update_attr = :update_val",
            ConditionExpression="#cond_attr = :cond_val",
            ExpressionAttributeNames={
                "#update_attr": update_attribute,
                "#cond_attr": condition_attribute,
            },
            ExpressionAttributeValues={":update_val": update_value, ":cond_val":
 condition_value},
            ReturnValues="UPDATED_NEW",
        )
        # Update succeeded, condition was met
        return True, response
    except ClientError as e:
        if e.response["Error"]["Code"] == "ConditionalCheckFailedException":
            # Condition was not met
            return False, None
        else:
            # Other error occurred
            raise


def conditional_delete(
    table_name: str, key: Dict[str, Any], condition_attribute: str,
 condition_value: Any
) -> bool:
    """
    Delete an item only if a condition is met.

    This function demonstrates how to perform a conditional delete operation
    and determine if the condition was met.

    Args:
        table_name (str): The name of the DynamoDB table.
```

```
        key (Dict[str, Any]): The primary key of the item to delete.
        condition_attribute (str): The attribute to check in the condition.
        condition_value (Any): The value to compare against.

    Returns:
        bool: True if the delete succeeded (condition was met), False otherwise.
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    try:
        # Perform the conditional delete
        table.delete_item(
            Key=key,
            ConditionExpression="#attr = :val",
            ExpressionAttributeNames={"#attr": condition_attribute},
            ExpressionAttributeValues={":val": condition_value},
        )
        # Delete succeeded, condition was met
        return True
    except ClientError as e:
        if e.response["Error"]["Code"] == "ConditionalCheckFailedException":
            # Condition was not met
            return False
        else:
            # Other error occurred
            raise


def optimistic_locking_update(
    table_name: str,
    key: Dict[str, Any],
    version_attribute: str,
    update_attribute: str,
    update_value: Any,
) -> Tuple[bool, Optional[Dict[str, Any]]]:
    """
    Update an item using optimistic locking with a version attribute.

    This function demonstrates how to implement optimistic locking using
    a version attribute that is incremented with each update.

    Args:
```

```
        table_name (str): The name of the DynamoDB table.
        key (Dict[str, Any]): The primary key of the item to update.
        version_attribute (str): The name of the version attribute.
        update_attribute (str): The attribute to update.
        update_value (Any): The new value to set.

    Returns:
        Tuple[bool, Optional[Dict[str, Any]]]: A tuple containing:
            - A boolean indicating if the update succeeded
            - The response from DynamoDB if successful, None otherwise
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # First, get the current version
    try:
        response = table.get_item(
            Key=key,
            ProjectionExpression=f"#{version_attribute}",
            ExpressionAttributeNames={f"#{version_attribute}":
version_attribute},
        )

        item = response.get("Item", {})
        current_version = item.get(version_attribute, 0)

        # Now, try to update with a condition on the version
        try:
            update_response = table.update_item(
                Key=key,
                UpdateExpression=f"SET #{update_attribute} = :update_val,
#{version_attribute} = :new_version",
                ConditionExpression=f"#{version_attribute} = :current_version",
                ExpressionAttributeNames={
                    f"#{update_attribute}": update_attribute,
                    f"#{version_attribute}": version_attribute,
                },
                ExpressionAttributeValues={
                    ":update_val": update_value,
                    ":current_version": current_version,
                    ":new_version": current_version + 1,
                },
                ReturnValues="UPDATED_NEW",
```

```
            )
            # Update succeeded
            return True, update_response
        except ClientError as e:
            if e.response["Error"]["Code"] == "ConditionalCheckFailedException":
                # Version has changed, optimistic locking failed
                return False, None
            else:
                # Other error occurred
                raise
    except ClientError:
        # Error getting the item
        raise


def conditional_check_and_update(
    table_name: str,
    key: Dict[str, Any],
    check_attribute: str,
    check_value: Any,
    update_attribute: str,
    update_value: Any,
    create_if_not_exists: bool = False,
) -> Union[Dict[str, Any], None]:
    """
    Check if an attribute has a specific value and update another attribute if it
 does.

    This function demonstrates a more complex conditional update that can also
    create the item if it doesn't exist.

    Args:
        table_name (str): The name of the DynamoDB table.
        key (Dict[str, Any]): The primary key of the item to update.
        check_attribute (str): The attribute to check in the condition.
        check_value (Any): The value to compare against.
        update_attribute (str): The attribute to update.
        update_value (Any): The new value to set.
        create_if_not_exists (bool, optional): Whether to create the item if it
 doesn't exist.

    Returns:
        Union[Dict[str, Any], None]: The response from DynamoDB if successful,
 None otherwise.
```

```
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    try:
        if create_if_not_exists:
            # Use attribute_not_exists to create the item if it doesn't exist
            condition_expression = "attribute_not_exists(#pk) OR #check_attr
= :check_val"
            update_expression = "SET #update_attr = :update_val, #check_attr =
if_not_exists(#check_attr, :check_val)"

            # Get the partition key name from the key dictionary
            pk_name = next(iter(key))

            expression_attribute_names = {
                "#pk": pk_name,
                "#check_attr": check_attribute,
                "#update_attr": update_attribute,
            }
        else:
            # Only update if the check attribute has the expected value
            condition_expression = "#check_attr = :check_val"
            update_expression = "SET #update_attr = :update_val"

            expression_attribute_names = {
                "#check_attr": check_attribute,
                "#update_attr": update_attribute,
            }

        # Perform the conditional update
        response = table.update_item(
            Key=key,
            UpdateExpression=update_expression,
            ConditionExpression=condition_expression,
            ExpressionAttributeNames=expression_attribute_names,
            ExpressionAttributeValues={":check_val": check_value, ":update_val":
update_value},
            ReturnValues="UPDATED_NEW",
        )
        return response
    except ClientError as e:
        if e.response["Error"]["Code"] == "ConditionalCheckFailedException":
```

```
            # Condition was not met
            return None
        else:
            # Other error occurred
            raise
```

Example usage of conditional operations with AWS SDK for Python (Boto3).

```
def example_usage():
    """Example of how to use the conditional operations functions."""
    # Example parameters
    table_name = "Products"
    key = {"ProductId": "prod123"}

    print("Example 1: Conditional Update")
    try:
        # Update the price only if the current stock is greater than 10
        success, response = conditional_update(
            table_name=table_name,
            key=key,
            condition_attribute="Stock",
            condition_value=10,
            update_attribute="Price",
            update_value=99.99,
        )

        if success:
            # Fix for mypy: Handle the case where response might be None
            attributes = {} if response is None else response.get("Attributes",
 {})
            print(f"Update succeeded! New values: {attributes}")
        else:
            print("Update failed because the condition was not met.")
    except Exception as e:
        print(f"Error during conditional update: {e}")

    print("\nExample 2: Conditional Delete")
    try:
        # Delete the product only if it's discontinued
        success = conditional_delete(
```

```
            table_name=table_name,
            key=key,
            condition_attribute="Status",
            condition_value="Discontinued",
        )

        if success:
            print("Delete succeeded! The item was deleted.")
        else:
            print("Delete failed because the condition was not met.")
    except Exception as e:
        print(f"Error during conditional delete: {e}")

    print("\nExample 3: Optimistic Locking")
    try:
        # Update with optimistic locking using a version attribute
        success, response = optimistic_locking_update(
            table_name=table_name,
            key=key,
            version_attribute="Version",
            update_attribute="Description",
            update_value="Updated product description",
        )

        if success:
            # Fix for mypy: Handle the case where response might be None
            attributes = {} if response is None else response.get("Attributes",
{})
            print(f"Optimistic locking update succeeded! New values:
{attributes}")
        else:
            print("Optimistic locking update failed because the version has
changed.")
    except Exception as e:
        print(f"Error during optimistic locking update: {e}")

    print("\nExample 4: Conditional Check and Update")
    try:
        # Update the featured status if the product is in stock
        response = conditional_check_and_update(
            table_name=table_name,
            key=key,
            check_attribute="InStock",
            check_value=True,
```

```
            update_attribute="Featured",
            update_value=True,
            create_if_not_exists=True,
        )

        if response:
            print(
                f"Conditional check and update succeeded! New values:
{response.get('Attributes', {})}"
            )
        else:
            print("Conditional check and update failed because the condition was
not met.")
    except Exception as e:
        print(f"Error during conditional check and update: {e}")

    print("\nUnderstanding Conditional Operations in DynamoDB:")
    print("1. Conditional operations help maintain data integrity")
    print("2. They prevent race conditions in concurrent environments")
    print("3. Failed conditions result in ConditionalCheckFailedException")
    print("4. No DynamoDB capacity is consumed when conditions fail")
    print("5. Optimistic locking is a common pattern using version attributes")
    print("6. Conditions can be combined with logical operators (AND, OR, NOT)")
    print("7. Conditions can use comparison operators (=, <>, <, <=, >, >=)")
    print(
        "8. attribute_exists() and attribute_not_exists() are useful for checking
attribute presence"
    )
```

- For API details, see the following topics in *AWS SDK for Python (Boto3) API Reference*.
  - DeleteItem
  - PutItem
  - UpdateItem

For a complete list of AWS SDK developer guides and code examples, see Using DynamoDB with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use expression attribute names in DynamoDB with an AWS SDK

The following code examples show how to use expression attribute names in DynamoDB.

- Work with reserved words in DynamoDB expressions.
- Use expression attribute name placeholders.
- Handle special characters in attribute names.

Java

SDK for Java 2.x

Demonstrate expression attribute names using AWS SDK for Java 2.x.

```
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.QueryRequest;
import software.amazon.awssdk.services.dynamodb.model.QueryResponse;
import software.amazon.awssdk.services.dynamodb.model.ScanRequest;
import software.amazon.awssdk.services.dynamodb.model.ScanResponse;
import software.amazon.awssdk.services.dynamodb.model.UpdateItemRequest;
import software.amazon.awssdk.services.dynamodb.model.UpdateItemResponse;

import java.util.Arrays;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

    /**
     * Updates an attribute that is a reserved word in DynamoDB.
     *
     * <p>This method demonstrates how to use expression attribute names to
 update
     * attributes that are reserved words in DynamoDB.
     *
     * @param dynamoDbClient The DynamoDB client
     * @param tableName The name of the DynamoDB table
     * @param key The key of the item to update
     * @param reservedWordAttribute The reserved word attribute to update
     * @param value The value to set
     * @return The response from DynamoDB
```

```
     * @throws DynamoDbException if an error occurs during the operation
     */
    public static UpdateItemResponse updateReservedWordAttribute(
        DynamoDbClient dynamoDbClient,
        String tableName,
        Map<String, AttributeValue> key,
        String reservedWordAttribute,
        AttributeValue value) {

        // Define the update parameters using expression attribute names
        UpdateItemRequest request = UpdateItemRequest.builder()
            .tableName(tableName)
            .key(key)
            .updateExpression("SET #attr = :value")
            .expressionAttributeNames(Map.of("#attr", reservedWordAttribute))
            .expressionAttributeValues(Map.of(":value", value))
            .returnValues("UPDATED_NEW")
            .build();

        // Perform the update operation
        return dynamoDbClient.updateItem(request);
    }

    /**
     * Updates an attribute that contains special characters.
     *
     * <p>This method demonstrates how to use expression attribute names to
update
     * attributes that contain special characters.
     *
     * @param dynamoDbClient The DynamoDB client
     * @param tableName The name of the DynamoDB table
     * @param key The key of the item to update
     * @param specialCharAttribute The attribute with special characters to
update
     * @param value The value to set
     * @return The response from DynamoDB
     * @throws DynamoDbException if an error occurs during the operation
     */
    public static UpdateItemResponse updateSpecialCharacterAttribute(
        DynamoDbClient dynamoDbClient,
        String tableName,
        Map<String, AttributeValue> key,
        String specialCharAttribute,
```

```
        AttributeValue value) {

        // Define the update parameters using expression attribute names
        UpdateItemRequest request = UpdateItemRequest.builder()
            .tableName(tableName)
            .key(key)
            .updateExpression("SET #attr = :value")
            .expressionAttributeNames(Map.of("#attr", specialCharAttribute))
            .expressionAttributeValues(Map.of(":value", value))
            .returnValues("UPDATED_NEW")
            .build();

        // Perform the update operation
        return dynamoDbClient.updateItem(request);
    }

    /**
     * Queries items using an attribute that is a reserved word.
     *
     * <p>This method demonstrates how to use expression attribute names in a
query
     * when the attribute is a reserved word.
     *
     * @param dynamoDbClient The DynamoDB client
     * @param tableName The name of the DynamoDB table
     * @param partitionKeyName The name of the partition key attribute
     * @param partitionKeyValue The value of the partition key
     * @param reservedWordAttribute The reserved word attribute to filter on
     * @param value The value to compare against
     * @return The response from DynamoDB
     * @throws DynamoDbException if an error occurs during the operation
     */
    public static QueryResponse queryWithReservedWordAttribute(
        DynamoDbClient dynamoDbClient,
        String tableName,
        String partitionKeyName,
        AttributeValue partitionKeyValue,
        String reservedWordAttribute,
        AttributeValue value) {

        // Define the query parameters using expression attribute names
        Map<String, String> expressionAttributeNames = new HashMap<>();
        expressionAttributeNames.put("#pkName", partitionKeyName);
        expressionAttributeNames.put("#attr", reservedWordAttribute);
```

```
        Map<String, AttributeValue> expressionAttributeValues = new HashMap<>();
        expressionAttributeValues.put(":pkValue", partitionKeyValue);
        expressionAttributeValues.put(":value", value);

        QueryRequest request = QueryRequest.builder()
            .tableName(tableName)
            .keyConditionExpression("#pkName = :pkValue")
            .filterExpression("#attr = :value")
            .expressionAttributeNames(expressionAttributeNames)
            .expressionAttributeValues(expressionAttributeValues)
            .build();

        // Perform the query operation
        return dynamoDbClient.query(request);
    }

    /**
     * Updates a nested attribute with a path that contains reserved words.
     *
     * <p>This method demonstrates how to use expression attribute names to
update
     * nested attributes where the path contains reserved words.
     *
     * @param dynamoDbClient The DynamoDB client
     * @param tableName The name of the DynamoDB table
     * @param key The key of the item to update
     * @param attributePath The path to the nested attribute as an array
     * @param value The value to set
     * @return The response from DynamoDB
     * @throws DynamoDbException if an error occurs during the operation
     */
    public static UpdateItemResponse updateNestedReservedWordAttribute(
        DynamoDbClient dynamoDbClient,
        String tableName,
        Map<String, AttributeValue> key,
        List<String> attributePath,
        AttributeValue value) {

        // Create expression attribute names for each part of the path
        Map<String, String> expressionAttributeNames = new HashMap<>();
        for (int i = 0; i < attributePath.size(); i++) {
            expressionAttributeNames.put("#attr" + i, attributePath.get(i));
        }
```

```
        // Build the attribute path using the expression attribute names
        StringBuilder attributePathExpression = new StringBuilder();
        for (int i = 0; i < attributePath.size(); i++) {
            if (i > 0) {
                attributePathExpression.append(".");
            }
            attributePathExpression.append("#attr").append(i);
        }

        // Define the update parameters
        UpdateItemRequest request = UpdateItemRequest.builder()
            .tableName(tableName)
            .key(key)
            .updateExpression("SET " + attributePathExpression.toString() + "
= :value")
            .expressionAttributeNames(expressionAttributeNames)
            .expressionAttributeValues(Map.of(":value", value))
            .returnValues("UPDATED_NEW")
            .build();

        // Perform the update operation
        return dynamoDbClient.updateItem(request);
    }

    /**
     * Scans a table with multiple attribute name placeholders.
     *
     * <p>This method demonstrates how to use multiple expression attribute names
     * in a complex filter expression.
     *
     * @param dynamoDbClient The DynamoDB client
     * @param tableName The name of the DynamoDB table
     * @param filters Object mapping attribute names to filter values
     * @return The response from DynamoDB
     * @throws DynamoDbException if an error occurs during the operation
     */
    public static ScanResponse scanWithMultipleAttributeNames(
        DynamoDbClient dynamoDbClient, String tableName, Map<String,
AttributeValue> filters) {

        // Create expression attribute names and values
        Map<String, String> expressionAttributeNames = new HashMap<>();
        Map<String, AttributeValue> expressionAttributeValues = new HashMap<>();
```

```
        StringBuilder filterExpression = new StringBuilder();

        // Build the filter expression
        int index = 0;
        for (Map.Entry<String, AttributeValue> entry : filters.entrySet()) {
            String attrName = entry.getKey();
            AttributeValue attrValue = entry.getValue();

            String nameKey = "#attr" + index;
            String valueKey = ":val" + index;

            expressionAttributeNames.put(nameKey, attrName);
            expressionAttributeValues.put(valueKey, attrValue);

            // Add AND between conditions (except for the first one)
            if (index > 0) {
                filterExpression.append(" AND ");
            }

            filterExpression.append(nameKey).append(" = ").append(valueKey);
            index++;
        }

        // Define the scan parameters
        ScanRequest request = ScanRequest.builder()
            .tableName(tableName)
            .filterExpression(filterExpression.toString())
            .expressionAttributeNames(expressionAttributeNames)
            .expressionAttributeValues(expressionAttributeValues)
            .build();

        // Perform the scan operation
        return dynamoDbClient.scan(request);
    }
```

Example usage of expression attribute names with AWS SDK for Java 2.x.

```
    public static void exampleUsage(DynamoDbClient dynamoDbClient, String
tableName) {
        // Example key
        Map<String, AttributeValue> key = new HashMap<>();
        key.put("ProductId", AttributeValue.builder().s("P12345").build());
```

```
        System.out.println("Demonstrating expression attribute names in
DynamoDB");

        try {
            // Example 1: Update an attribute that is a reserved word
            System.out.println("\nExample 1: Updating an attribute that is a
reserved word");
            UpdateItemResponse response1 = updateReservedWordAttribute(
                dynamoDbClient,
                tableName,
                key,
                "Size", // "SIZE" is a reserved word in DynamoDB
                AttributeValue.builder().s("Large").build());

            System.out.println("Updated attribute: " + response1.attributes());

            // Example 2: Update an attribute with special characters
            System.out.println("\nExample 2: Updating an attribute with special
characters");
            UpdateItemResponse response2 = updateSpecialCharacterAttribute(
                dynamoDbClient,
                tableName,
                key,
                "Product-Type", // Contains a hyphen, which is a special
character
                AttributeValue.builder().s("Electronics").build());

            System.out.println("Updated attribute: " + response2.attributes());

            // Example 3: Query with a reserved word attribute
            System.out.println("\nExample 3: Querying with a reserved word
attribute");
            QueryResponse response3 = queryWithReservedWordAttribute(
                dynamoDbClient,
                tableName,
                "Category",
                AttributeValue.builder().s("Electronics").build(),
                "Count", // "COUNT" is a reserved word in DynamoDB
                AttributeValue.builder().n("10").build());

            System.out.println("Found " + response3.count() + " items");
```

```
            // Example 4: Update a nested attribute with reserved words in the
path
            System.out.println("\nExample 4: Updating a nested attribute with
reserved words in the path");
            UpdateItemResponse response4 = updateNestedReservedWordAttribute(
                dynamoDbClient,
                tableName,
                key,
                Arrays.asList("Dimensions", "Size", "Height"), // "SIZE" is a
reserved word
                AttributeValue.builder().n("30").build());

            System.out.println("Updated nested attribute: " +
response4.attributes());

            // Example 5: Scan with multiple attribute name placeholders
            System.out.println("\nExample 5: Scanning with multiple attribute
name placeholders");
            Map<String, AttributeValue> filters = new HashMap<>();
            filters.put("Size", AttributeValue.builder().s("Large").build());
            filters.put("Count", AttributeValue.builder().n("10").build());
            filters.put(
                "Product-Type",
AttributeValue.builder().s("Electronics").build());

            ScanResponse response5 =
scanWithMultipleAttributeNames(dynamoDbClient, tableName, filters);

            System.out.println("Found " + response5.count() + " items");

            // Show some common reserved words
            System.out.println("\nSome common DynamoDB reserved words:");
            List<String> commonReservedWords = getDynamoDBReservedWords();
            System.out.println(String.join(", ", commonReservedWords));

            // Explain expression attribute names
            System.out.println("\nKey points about expression attribute names:");
            System.out.println("1. Use expression attribute names (#name) for
reserved words");
            System.out.println("2. Use expression attribute names for attributes
with special characters");
            System.out.println(
                "3. Special characters include: spaces, hyphens, dots, and other
non-alphanumeric characters");
```

```
            System.out.println("4. Expression attribute names are required for
 nested attributes with reserved words");
            System.out.println("5. You can use multiple expression attribute
 names in a single expression");
            System.out.println("6. Expression attribute names are case-
sensitive");
            System.out.println("7. Expression attribute names are only used in
 expressions, not in the actual data");

        } catch (DynamoDbException e) {
            System.err.println("Error: " + e.getMessage());
            e.printStackTrace();
        }
    }
```

- For API details, see the following topics in *AWS SDK for Java 2.x API Reference*.
  - Query
  - UpdateItem

JavaScript

SDK for JavaScript (v3)

Demonstrate expression attribute names using AWS SDK for JavaScript.

```
const { DynamoDBClient } = require("@aws-sdk/client-dynamodb");
const {
  DynamoDBDocumentClient,
  UpdateCommand,
  GetCommand,
  QueryCommand,
  ScanCommand
} = require("@aws-sdk/lib-dynamodb");

/**
 * Update an attribute that is a reserved word in DynamoDB.
 *
 * This function demonstrates how to use expression attribute names to update
 * attributes that are reserved words in DynamoDB.
 *
 * @param {Object} config - AWS configuration object
```

```
 * @param {string} tableName - The name of the DynamoDB table
 * @param {Object} key - The key of the item to update
 * @param {string} reservedWordAttribute - The reserved word attribute to update
 * @param {any} value - The value to set
 * @returns {Promise<Object>} - The response from DynamoDB
 */
async function updateReservedWordAttribute(
  config,
  tableName,
  key,
  reservedWordAttribute,
  value
) {
  // Initialize the DynamoDB client
  const client = new DynamoDBClient(config);
  const docClient = DynamoDBDocumentClient.from(client);

  // Define the update parameters using expression attribute names
  const params = {
    TableName: tableName,
    Key: key,
    UpdateExpression: "SET #attr = :value",
    ExpressionAttributeNames: {
      "#attr": reservedWordAttribute
    },
    ExpressionAttributeValues: {
      ":value": value
    },
    ReturnValues: "UPDATED_NEW"
  };

  // Perform the update operation
  const response = await docClient.send(new UpdateCommand(params));

  return response;
}

/**
 * Update an attribute that contains special characters.
 *
 * This function demonstrates how to use expression attribute names to update
 * attributes that contain special characters.
 *
 * @param {Object} config - AWS configuration object
```

```
 * @param {string} tableName - The name of the DynamoDB table
 * @param {Object} key - The key of the item to update
 * @param {string} specialCharAttribute - The attribute with special characters
 to update
 * @param {any} value - The value to set
 * @returns {Promise<Object>} - The response from DynamoDB
 */
async function updateSpecialCharacterAttribute(
  config,
  tableName,
  key,
  specialCharAttribute,
  value
) {
  // Initialize the DynamoDB client
  const client = new DynamoDBClient(config);
  const docClient = DynamoDBDocumentClient.from(client);

  // Define the update parameters using expression attribute names
  const params = {
    TableName: tableName,
    Key: key,
    UpdateExpression: "SET #attr = :value",
    ExpressionAttributeNames: {
      "#attr": specialCharAttribute
    },
    ExpressionAttributeValues: {
      ":value": value
    },
    ReturnValues: "UPDATED_NEW"
  };

  // Perform the update operation
  const response = await docClient.send(new UpdateCommand(params));

  return response;
}

/**
 * Query items using an attribute that is a reserved word.
 *
 * This function demonstrates how to use expression attribute names in a query
 * when the attribute is a reserved word.
 *
```

```
 * @param {Object} config - AWS configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {string} partitionKeyName - The name of the partition key attribute
 * @param {any} partitionKeyValue - The value of the partition key
 * @param {string} reservedWordAttribute - The reserved word attribute to filter
 on
 * @param {any} value - The value to compare against
 * @returns {Promise<Object>} - The response from DynamoDB
 */
async function queryWithReservedWordAttribute(
  config,
  tableName,
  partitionKeyName,
  partitionKeyValue,
  reservedWordAttribute,
  value
) {
  // Initialize the DynamoDB client
  const client = new DynamoDBClient(config);
  const docClient = DynamoDBDocumentClient.from(client);

  // Define the query parameters using expression attribute names
  const params = {
    TableName: tableName,
    KeyConditionExpression: "#pkName = :pkValue",
    FilterExpression: "#attr = :value",
    ExpressionAttributeNames: {
      "#pkName": partitionKeyName,
      "#attr": reservedWordAttribute
    },
    ExpressionAttributeValues: {
      ":pkValue": partitionKeyValue,
      ":value": value
    }
  };

  // Perform the query operation
  const response = await docClient.send(new QueryCommand(params));

  return response;
}

/**
 * Update a nested attribute with a path that contains reserved words.
```

```
 *
 * This function demonstrates how to use expression attribute names to update
 * nested attributes where the path contains reserved words.
 *
 * @param {Object} config - AWS configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {Object} key - The key of the item to update
 * @param {string[]} attributePath - The path to the nested attribute as an array
 * @param {any} value - The value to set
 * @returns {Promise<Object>} - The response from DynamoDB
 */
async function updateNestedReservedWordAttribute(
  config,
  tableName,
  key,
  attributePath,
  value
) {
  // Initialize the DynamoDB client
  const client = new DynamoDBClient(config);
  const docClient = DynamoDBDocumentClient.from(client);

  // Create expression attribute names for each part of the path
  const expressionAttributeNames = {};
  for (let i = 0; i < attributePath.length; i++) {
    expressionAttributeNames[`#attr${i}`] = attributePath[i];
  }

  // Build the attribute path using the expression attribute names
  const attributePathExpression = attributePath
    .map((_, i) => `#attr${i}`)
    .join(".");

  // Define the update parameters
  const params = {
    TableName: tableName,
    Key: key,
    UpdateExpression: `SET ${attributePathExpression} = :value`,
    ExpressionAttributeNames: expressionAttributeNames,
    ExpressionAttributeValues: {
      ":value": value
    },
    ReturnValues: "UPDATED_NEW"
  };
```

```
  // Perform the update operation
  const response = await docClient.send(new UpdateCommand(params));

  return response;
}

/**
 * Scan a table with multiple attribute name placeholders.
 *
 * This function demonstrates how to use multiple expression attribute names
 * in a complex filter expression.
 *
 * @param {Object} config - AWS configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {Object} filters - Object mapping attribute names to filter values
 * @returns {Promise<Object>} - The response from DynamoDB
 */
async function scanWithMultipleAttributeNames(
  config,
  tableName,
  filters
) {
  // Initialize the DynamoDB client
  const client = new DynamoDBClient(config);
  const docClient = DynamoDBDocumentClient.from(client);

  // Create expression attribute names and values
  const expressionAttributeNames = {};
  const expressionAttributeValues = {};
  const filterConditions = [];

  // Build the filter expression
  Object.entries(filters).forEach(([attrName, value], index) => {
    const nameKey = `#attr${index}`;
    const valueKey = `:val${index}`;

    expressionAttributeNames[nameKey] = attrName;
    expressionAttributeValues[valueKey] = value;
    filterConditions.push(`${nameKey} = ${valueKey}`);
  });

  // Join the filter conditions with AND
  const filterExpression = filterConditions.join(" AND ");
```

```
  // Define the scan parameters
  const params = {
    TableName: tableName,
    FilterExpression: filterExpression,
    ExpressionAttributeNames: expressionAttributeNames,
    ExpressionAttributeValues: expressionAttributeValues
  };

  // Perform the scan operation
  const response = await docClient.send(new ScanCommand(params));

  return response;
}

/**
 * Get the current value of an item.
 *
 * Helper function to retrieve the current value of an item.
 *
 * @param {Object} config - AWS configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {Object} key - The key of the item to get
 * @returns {Promise<Object|null>} - The item or null if not found
 */
async function getItem(
  config,
  tableName,
  key
) {
  // Initialize the DynamoDB client
  const client = new DynamoDBClient(config);
  const docClient = DynamoDBDocumentClient.from(client);

  // Define the get parameters
  const params = {
    TableName: tableName,
    Key: key
  };

  // Perform the get operation
  const response = await docClient.send(new GetCommand(params));

  // Return the item if it exists, otherwise null
```

```
  return response.Item || null;
}
```

Example usage of expression attribute names with AWS SDK for JavaScript.

```
/**
 * Example of how to use expression attribute names.
 */
async function exampleUsage() {
  // Example parameters
  const config = { region: "us-west-2" };
  const tableName = "Products";
  const key = { ProductId: "P12345" };

  console.log("Demonstrating expression attribute names in DynamoDB");

  try {
    // Example 1: Update an attribute that is a reserved word
    console.log("\nExample 1: Updating an attribute that is a reserved word");
    const response1 = await updateReservedWordAttribute(
      config,
      tableName,
      key,
      "Size", // "SIZE" is a reserved word in DynamoDB
      "Large"
    );

    console.log("Updated attribute:", response1.Attributes);

    // Example 2: Update an attribute with special characters
    console.log("\nExample 2: Updating an attribute with special characters");
    const response2 = await updateSpecialCharacterAttribute(
      config,
      tableName,
      key,
      "Product-Type", // Contains a hyphen, which is a special character
      "Electronics"
    );

    console.log("Updated attribute:", response2.Attributes);

    // Example 3: Query with a reserved word attribute
```

```
    console.log("\nExample 3: Querying with a reserved word attribute");
    const response3 = await queryWithReservedWordAttribute(
      config,
      tableName,
      "Category",
      "Electronics",
      "Count", // "COUNT" is a reserved word in DynamoDB
      10
    );

    console.log(`Found ${response3.Items.length} items`);

    // Example 4: Update a nested attribute with reserved words in the path
    console.log("\nExample 4: Updating a nested attribute with reserved words in
the path");
    const response4 = await updateNestedReservedWordAttribute(
      config,
      tableName,
      key,
      ["Dimensions", "Size", "Height"], // "SIZE" is a reserved word
      30
    );

    console.log("Updated nested attribute:", response4.Attributes);

    // Example 5: Scan with multiple attribute name placeholders
    console.log("\nExample 5: Scanning with multiple attribute name
placeholders");
    const response5 = await scanWithMultipleAttributeNames(
      config,
      tableName,
      {
        "Size": "Large",
        "Count": 10,
        "Product-Type": "Electronics"
      }
    );

    console.log(`Found ${response5.Items.length} items`);

    // Get the final state of the item
    console.log("\nFinal state of the item:");
    const item = await getItem(config, tableName, key);
    console.log(JSON.stringify(item, null, 2));
```

```
    // Show some common reserved words
    console.log("\nSome common DynamoDB reserved words:");
    const commonReservedWords = [
      "ABORT", "ABSOLUTE", "ACTION", "ADD", "ALL", "ALTER", "AND", "ANY", "AS",
      "ASC", "BETWEEN", "BY", "CASE", "CAST", "COLUMN", "CONNECT", "COUNT",
      "CREATE", "CURRENT", "DATE", "DELETE", "DESC", "DROP", "ELSE", "EXISTS",
      "FOR", "FROM", "GRANT", "GROUP", "HAVING", "IN", "INDEX", "INSERT", "INTO",
      "IS", "JOIN", "KEY", "LEVEL", "LIKE", "LIMIT", "LOCAL", "MAX", "MIN",
 "NAME",
      "NOT", "NULL", "OF", "ON", "OR", "ORDER", "OUTER", "REPLACE", "RETURN",
      "SELECT", "SET", "SIZE", "TABLE", "THEN", "TO", "UPDATE", "USER", "VALUES",
      "VIEW", "WHERE"
    ];
    console.log(commonReservedWords.join(", "));

    // Explain expression attribute names
    console.log("\nKey points about expression attribute names:");
    console.log("1. Use expression attribute names (#name) for reserved words");
    console.log("2. Use expression attribute names for attributes with special
 characters");
    console.log("3. Special characters include: spaces, hyphens, dots, and other
 non-alphanumeric characters");
    console.log("4. Expression attribute names are required for nested attributes
 with reserved words");
    console.log("5. You can use multiple expression attribute names in a single
 expression");
    console.log("6. Expression attribute names are case-sensitive");
    console.log("7. Expression attribute names are only used in expressions, not
 in the actual data");

  } catch (error) {
    console.error("Error:", error);
  }
}
```

- For API details, see the following topics in *AWS SDK for JavaScript API Reference*.
  - Query
  - UpdateItem

Python

SDK for Python (Boto3)

Demonstrate expression attribute names using AWS SDK for Python (Boto3).

```
import boto3
from botocore.exceptions import ClientError
from typing import Any, Dict, List


def use_reserved_word_attribute(
    table_name: str, key: Dict[str, Any], reserved_word: str, value: Any
) -> Dict[str, Any]:
    """
    Update an attribute whose name is a DynamoDB reserved word.

    This function demonstrates how to use expression attribute names to work with
    attributes that have names that are DynamoDB reserved words.

    Args:
        table_name (str): The name of the DynamoDB table.
        key (Dict[str, Any]): The primary key of the item to update.
        reserved_word (str): The reserved word to use as an attribute name.
        value (Any): The value to set for the attribute.

    Returns:
        Dict[str, Any]: The response from DynamoDB containing the updated
 attribute values.
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # Use expression attribute names to handle the reserved word
    response = table.update_item(
        Key=key,
        UpdateExpression="SET #reserved_attr = :value",
        ExpressionAttributeNames={"#reserved_attr": reserved_word},
        ExpressionAttributeValues={":value": value},
        ReturnValues="UPDATED_NEW",
    )

    return response
```

```
def use_special_character_attribute(
    table_name: str, key: Dict[str, Any], attribute_with_special_chars: str,
 value: Any
) -> Dict[str, Any]:
    """
    Update an attribute whose name contains special characters.

    This function demonstrates how to use expression attribute names to work with
    attributes that have names containing special characters like spaces, dots,
 or hyphens.

    Args:
        table_name (str): The name of the DynamoDB table.
        key (Dict[str, Any]): The primary key of the item to update.
        attribute_with_special_chars (str): The attribute name with special
 characters.
        value (Any): The value to set for the attribute.

    Returns:
        Dict[str, Any]: The response from DynamoDB containing the updated
 attribute values.
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # Use expression attribute names to handle special characters
    response = table.update_item(
        Key=key,
        UpdateExpression="SET #special_attr = :value",
        ExpressionAttributeNames={"#special_attr": attribute_with_special_chars},
        ExpressionAttributeValues={":value": value},
        ReturnValues="UPDATED_NEW",
    )

    return response


def query_with_attribute_names(
    table_name: str,
    partition_key_name: str,
    partition_key_value: str,
```

```
    filter_attribute_name: str,
    filter_value: Any,
) -> Dict[str, Any]:
    """
    Query a table using expression attribute names for both key and filter
 attributes.

    This function demonstrates how to use expression attribute names in a query
 operation
    for both the key condition expression and filter expression.

    Args:
        table_name (str): The name of the DynamoDB table.
        partition_key_name (str): The name of the partition key attribute.
        partition_key_value (str): The value of the partition key to query.
        filter_attribute_name (str): The name of the attribute to filter on.
        filter_value (Any): The value to compare against in the filter.

    Returns:
        Dict[str, Any]: The response from DynamoDB containing the query results.
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # Use expression attribute names for both key condition and filter
    response = table.query(
        KeyConditionExpression="#pk = :pk_val",
        FilterExpression="#filter_attr = :filter_val",
        ExpressionAttributeNames={"#pk": partition_key_name, "#filter_attr":
 filter_attribute_name},
        ExpressionAttributeValues={":pk_val": partition_key_value, ":filter_val":
 filter_value},
    )

    return response


def update_nested_attribute_with_dots(
    table_name: str, key: Dict[str, Any], path_with_dots: str, value: Any
) -> Dict[str, Any]:
    """
    Update a nested attribute using a path with dot notation.
```

```
    This function demonstrates how to use expression attribute names to work with
    nested attributes specified using dot notation.

    Args:
        table_name (str): The name of the DynamoDB table.
        key (Dict[str, Any]): The primary key of the item to update.
        path_with_dots (str): The path to the nested attribute using dot notation
(e.g., "a.b.c").
        value (Any): The value to set for the nested attribute.

    Returns:
        Dict[str, Any]: The response from DynamoDB containing the updated
attribute values.
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # Split the path into components
    path_parts = path_with_dots.split(".")

    # Build the update expression and attribute names
    update_expression = "SET "
    expression_attribute_names = {}

    # Build the path expression
    path_expression = ""
    for i, part in enumerate(path_parts):
        name_placeholder = f"#attr{i}"
        expression_attribute_names[name_placeholder] = part

        if i == 0:
            path_expression = name_placeholder
        else:
            path_expression += f".{name_placeholder}"

    # Complete the update expression
    update_expression += f"{path_expression} = :value"

    # Execute the update
    response = table.update_item(
        Key=key,
        UpdateExpression=update_expression,
        ExpressionAttributeNames=expression_attribute_names,
```

```
        ExpressionAttributeValues={":value": value},
        ReturnValues="UPDATED_NEW",
    )

    return response


def demonstrate_attribute_name_requirements(table_name: str, key: Dict[str, Any])
 -> Dict[str, Any]:
    """
    Demonstrate the requirements and allowed characters for attribute names.

    This function shows examples of valid and invalid attribute names and how to
    handle them using expression attribute names.

    Args:
        table_name (str): The name of the DynamoDB table.
        key (Dict[str, Any]): The primary key of the item to update.

    Returns:
        Dict[str, Any]: A dictionary containing the results of the demonstration.
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # Examples of attribute names with different characteristics
    examples = {
        "valid_standard": "NormalAttribute",  # Standard attribute name (no
 placeholder needed)
        "valid_with_underscore": "Normal_Attribute",  # Underscore is allowed
        "valid_with_number": "Attribute123",  # Numbers are allowed
        "reserved_word": "Timestamp",  # Reserved word (requires placeholder)
        "starts_with_number": "123Attribute",  # Starts with number (valid but
 may need placeholder in some contexts)
        "with_space": "Attribute Name",  # Contains space (requires placeholder)
        "with_dot": "Attribute.Name",  # Contains dot (requires placeholder)
        "with_hyphen": "Attribute-Name",  # Contains hyphen (requires
 placeholder)
        "with_special_chars": "Attribute#$%",  # Contains special characters
 (requires placeholder)
    }

    results = {}
```

```
   # Try to update each attribute type
   for example_type, attr_name in examples.items():
       try:
           # For attributes that don't need placeholders, try direct reference
           if example_type in ["valid_standard", "valid_with_underscore",
"valid_with_number"]:
               try:
                   # Try without expression attribute names first
                   response = table.update_item(
                       Key=key,
                       UpdateExpression=f"SET {attr_name} = :value",
                       ExpressionAttributeValues={":value": f"Value for
{attr_name}"},
                       ReturnValues="UPDATED_NEW",
                   )
                   results[example_type] = {
                       "attribute_name": attr_name,
                       "success": True,
                       "needed_placeholder": False,
                       "response": response,
                   }
               except ClientError:
                   # If direct reference fails, try with placeholder
                   response = table.update_item(
                       Key=key,
                       UpdateExpression="SET #attr = :value",
                       ExpressionAttributeNames={"#attr": attr_name},
                       ExpressionAttributeValues={":value": f"Value for
{attr_name}"},
                       ReturnValues="UPDATED_NEW",
                   )
                   results[example_type] = {
                       "attribute_name": attr_name,
                       "success": True,
                       "needed_placeholder": True,
                       "response": response,
                   }
           else:
               # For attributes that definitely need placeholders
               response = table.update_item(
                   Key=key,
                   UpdateExpression="SET #attr = :value",
                   ExpressionAttributeNames={"#attr": attr_name},
```

```
                    ExpressionAttributeValues={":value": f"Value for
 {attr_name}"},
                    ReturnValues="UPDATED_NEW",
                )
                results[example_type] = {
                    "attribute_name": attr_name,
                    "success": True,
                    "needed_placeholder": True,
                    "response": response,
                }
        except ClientError as e:
            results[example_type] = {"attribute_name": attr_name, "success":
 False, "error": str(e)}

    return results
```

Example usage of expression attribute names with AWS SDK for Python (Boto3).

```
def example_usage():
    """Example of how to use expression attribute names in DynamoDB."""
    # Example parameters
    table_name = "Products"
    key = {"ProductId": "prod123"}

    print("Example 1: Using a reserved word as an attribute name")
    try:
        response = use_reserved_word_attribute(
            table_name=table_name, key=key, reserved_word="Timestamp",
 value="2025-05-14T12:00:00Z"
        )
        print(f"Reserved word attribute updated successfully:
 {response.get('Attributes', {})}")
    except Exception as e:
        print(f"Error updating reserved word attribute: {e}")

    print("\nExample 2: Using an attribute name with special characters")
    try:
        response = use_special_character_attribute(
            table_name=table_name,
            key=key,
```

```
            attribute_with_special_chars="Product Info",
            value="Special product information",
        )
        print(f"Special character attribute updated successfully:
{response.get('Attributes', {})}")
    except Exception as e:
        print(f"Error updating special character attribute: {e}")

    print("\nExample 3: Querying with expression attribute names")
    try:
        response = query_with_attribute_names(
            table_name=table_name,
            partition_key_name="Category",
            partition_key_value="Electronics",
            filter_attribute_name="Price",
            filter_value=500,
        )
        print(
            f"Query with expression attribute names returned
{len(response.get('Items', []))} items"
        )
    except Exception as e:
        print(f"Error querying with expression attribute names: {e}")

    print("\nExample 4: Updating a nested attribute with dot notation")
    try:
        response = update_nested_attribute_with_dots(
            table_name=table_name,
            key=key,
            path_with_dots="Product.Details.Specifications",
            value={"Weight": "2.5 kg", "Dimensions": "30x20x10 cm"},
        )
        print(f"Nested attribute updated successfully:
{response.get('Attributes', {})}")
    except Exception as e:
        print(f"Error updating nested attribute: {e}")

    print("\nExample 5: Demonstrating attribute name requirements")
    try:
        results = demonstrate_attribute_name_requirements(table_name=table_name,
key=key)

        print("Attribute Name Requirements Results:")
        for example_type, result in results.items():
```

```
            if result.get("success", False):
                needed_placeholder = result.get("needed_placeholder", True)
                print(
                    f"  - {example_type}: '{result['attribute_name']}' -
{'Requires' if needed_placeholder else 'Does not require'} placeholder"
                )
            else:
                print(
                    f"  - {example_type}: '{result['attribute_name']}' - Failed:
{result.get('error', 'Unknown error')}"
                )
    except Exception as e:
        print(f"Error demonstrating attribute name requirements: {e}")

    print("\nCommon DynamoDB Reserved Words (sample):")
    reserved_words = get_common_reserved_words()
    print(", ".join(reserved_words[:20]) + "... (and many more)")

    print("\nWhen to Use Expression Attribute Names:")
    print("1. When the attribute name is a DynamoDB reserved word")
    print("2. When the attribute name contains special characters (spaces, dots,
hyphens)")
    print("3. When the attribute name begins with a number")
    print("4. When working with nested attributes using dot notation")
    print("5. When you need to reference the same attribute multiple times in an
expression")

    print("\nExpression Attribute Name Requirements:")
    print("1. Must begin with a pound sign (#)")
    print("2. After the pound sign, must contain at least one character")
    print("3. Can contain alphanumeric characters and underscore (_)")
    print("4. Are case-sensitive")
    print("5. Must be unique within a single expression")

    print("\nAttribute Name Requirements in DynamoDB:")
    print("1. Can begin with a-z, A-Z, or 0-9")
    print("2. Can contain a-z, A-Z, 0-9, underscore (_), dash (-), and dot (.)")
    print("3. Are case-sensitive")
    print("4. No length restrictions, but practical limits apply")
    print("5. Cannot be a DynamoDB reserved word if used directly in
expressions")
```

- For API details, see the following topics in *AWS SDK for Python (Boto3) API Reference*.
  - Query
  - UpdateItem

For a complete list of AWS SDK developer guides and code examples, see Using DynamoDB with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use scheduled events to invoke a Lambda function

The following code examples show how to create an AWS Lambda function invoked by an Amazon EventBridge scheduled event.

Java

SDK for Java 2.x

Shows how to create an Amazon EventBridge scheduled event that invokes an AWS Lambda function. Configure EventBridge to use a cron expression to schedule when the Lambda function is invoked. In this example, you create a Lambda function by using the Lambda Java runtime API. This example invokes different AWS services to perform a specific use case. This example demonstrates how to create an app that sends a mobile text message to your employees that congratulates them at the one year anniversary date.

For complete source code and instructions on how to set up and run, see the full example on GitHub.

Services used in this example

- CloudWatch Logs
- DynamoDB
- EventBridge
- Lambda
- Amazon SNS

JavaScript

SDK for JavaScript (v3)

Shows how to create an Amazon EventBridge scheduled event that invokes an AWS Lambda function. Configure EventBridge to use a cron expression to schedule when the Lambda function is invoked. In this example, you create a Lambda function by using the Lambda JavaScript runtime API. This example invokes different AWS services to perform a specific use case. This example demonstrates how to create an app that sends a mobile text message to your employees that congratulates them at the one year anniversary date.

For complete source code and instructions on how to set up and run, see the full example on GitHub.

This example is also available in the AWS SDK for JavaScript v3 developer guide.

Services used in this example

- CloudWatch Logs
- DynamoDB
- EventBridge
- Lambda
- Amazon SNS

Python

SDK for Python (Boto3)

This example shows how to register an AWS Lambda function as the target of a scheduled Amazon EventBridge event. The Lambda handler writes a friendly message and the full event data to Amazon CloudWatch Logs for later retrieval.

- Deploys a Lambda function.
- Creates an EventBridge scheduled event and makes the Lambda function the target.
- Grants permission to let EventBridge invoke the Lambda function.
- Prints the latest data from CloudWatch Logs to show the result of the scheduled invocations.
- Cleans up all resources created during the demo.

This example is best viewed on GitHub. For complete source code and instructions on how to set up and run, see the full example on GitHub.

Services used in this example

- CloudWatch Logs
- DynamoDB
- EventBridge
- Lambda
- Amazon SNS

For a complete list of AWS SDK developer guides and code examples, see Using DynamoDB with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Work with DynamoDB Local Secondary Indexes using AWS Command Line Interface v2

The following code example shows how to create and query tables with Local Secondary Indexes.

- Create a table with a Local Secondary Index (LSI).
- Create a table with multiple LSIs with different projection types.
- Query data using LSIs.

Bash

AWS CLI with Bash script

Create a table with a Local Secondary Index.

```
# Create a table with a Local Secondary Index
aws dynamodb create-table \
    --table-name CustomerOrders \
    --attribute-definitions \
        AttributeName=CustomerID,AttributeType=S \
        AttributeName=OrderID,AttributeType=S \
        AttributeName=OrderDate,AttributeType=S \
    --key-schema \
```

```
        AttributeName=CustomerID,KeyType=HASH \
        AttributeName=OrderID,KeyType=RANGE \
    --local-secondary-indexes \
        "IndexName=OrderDateIndex,\
        KeySchema=[{AttributeName=CustomerID,KeyType=HASH},
{AttributeName=OrderDate,KeyType=RANGE}],\
        Projection={ProjectionType=ALL}" \
    --billing-mode PAY_PER_REQUEST
```

Create a table with multiple LSIs.

```
# Create a table with multiple Local Secondary Indexes
aws dynamodb create-table \
    --table-name CustomerDetails \
    --attribute-definitions \
        AttributeName=CustomerID,AttributeType=S \
        AttributeName=Name,AttributeType=S \
        AttributeName=Email,AttributeType=S \
        AttributeName=RegistrationDate,AttributeType=S \
    --key-schema \
        AttributeName=CustomerID,KeyType=HASH \
        AttributeName=Name,KeyType=RANGE \
    --local-secondary-indexes \
        "[
            {
                \"IndexName\": \"EmailIndex\",
                \"KeySchema\": [
                    {\"AttributeName\":\"CustomerID\",\"KeyType\":\"HASH\"},
                    {\"AttributeName\":\"Email\",\"KeyType\":\"RANGE\"}
                ],
                \"Projection\": {\"ProjectionType\":\"INCLUDE\",
\"NonKeyAttributes\":[\"Address\",\"Phone\"]}
            },
            {
                \"IndexName\": \"RegistrationIndex\",
                \"KeySchema\": [
                    {\"AttributeName\":\"CustomerID\",\"KeyType\":\"HASH\"},
                    {\"AttributeName\":\"RegistrationDate\",\"KeyType\":\"RANGE
\"}
                ],
                \"Projection\": {\"ProjectionType\":\"KEYS_ONLY\"}
            }
```

```
        ]" \
    --billing-mode PAY_PER_REQUEST
```

Query data using LSIs.

```
# Query the OrderDateIndex LSI
aws dynamodb query \
    --table-name CustomerOrders \
    --index-name OrderDateIndex \
    --key-condition-expression "CustomerID = :custId AND OrderDate BETWEEN :date1
 AND :date2" \
    --expression-attribute-values '{
        ":custId": {"S": "C1"},
        ":date1": {"S": "2023-01-01"},
        ":date2": {"S": "2023-02-01"}
    }'

# Query with a filter expression
aws dynamodb query \
    --table-name CustomerOrders \
    --index-name OrderDateIndex \
    --key-condition-expression "CustomerID = :custId" \
    --filter-expression "Amount > :amount" \
    --expression-attribute-values '{
        ":custId": {"S": "C1"},
        ":amount": {"N": "150"}
    }'
```

- For API details, see the following topics in *AWS CLI Command Reference*.
  - CreateTable
  - Query

For a complete list of AWS SDK developer guides and code examples, see Using DynamoDB with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Work with DynamoDB Streams and Time-to-Live using AWS Command Line Interface v2

The following code example shows how to manage DynamoDB Streams and Time-to-Live features.

- Create a table with Streams enabled.
- Describe Streams.
- Create a Lambda function for processing Streams.
- Enable TTL on a table.
- Add items with TTL attributes.
- Describe TTL settings.

Bash

AWS CLI with Bash script

Create a table with Streams enabled.

```
# Create a table with DynamoDB Streams enabled
aws dynamodb create-table \
    --table-name StreamsDemo \
    --attribute-definitions \
        AttributeName=ID,AttributeType=S \
    --key-schema \
        AttributeName=ID,KeyType=HASH \
    --billing-mode PAY_PER_REQUEST \
    --stream-specification StreamEnabled=true,StreamViewType=NEW_AND_OLD_IMAGES
```

Describe Streams.

```
# Get information about the stream
aws dynamodb describe-table \
    --table-name StreamsDemo \
    --query "Table.StreamSpecification"

# Get the stream ARN
STREAM_ARN=$(aws dynamodb describe-table \
    --table-name StreamsDemo \
```

```
    --query "Table.LatestStreamArn" \
    --output text)

echo "Stream ARN: $STREAM_ARN"

# Describe the stream
aws dynamodbstreams describe-stream \
    --stream-arn $STREAM_ARN
```

Create a Lambda function for Streams.

```
# Step 1: Create an IAM role for the Lambda function
cat > trust-policy.json << 'EOF'
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "lambda.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
EOF

aws iam create-role \
    --role-name DynamoDBStreamsLambdaRole \
    --assume-role-policy-document file://trust-policy.json

# Step 2: Attach permissions to the role
aws iam attach-role-policy \
    --role-name DynamoDBStreamsLambdaRole \
    --policy-arn arn:aws:iam::aws:policy/service-role/
AWSLambdaDynamoDBExecutionRole

# Step 3: Create a Lambda function (code would be in a separate file)
echo "Lambda function creation would be done separately with appropriate code"

# Step 4: Create an event source mapping
echo "Example command to create event source mapping:"
```

```
echo "aws lambda create-event-source-mapping \\"
echo "    --function-name ProcessDynamoDBRecords \\"
echo "    --event-source $STREAM_ARN \\"
echo "    --batch-size 100 \\"
echo "    --starting-position LATEST"
```

Enable TTL on a table.

```
# Create a table for TTL demonstration
aws dynamodb create-table \
    --table-name TTLDemo \
    --attribute-definitions \
        AttributeName=ID,AttributeType=S \
    --key-schema \
        AttributeName=ID,KeyType=HASH \
    --billing-mode PAY_PER_REQUEST

# Wait for table to become active
aws dynamodb wait table-exists --table-name TTLDemo

# Enable TTL on the table
aws dynamodb update-time-to-live \
    --table-name TTLDemo \
    --time-to-live-specification "Enabled=true, AttributeName=ExpirationTime"
```

Add items with TTL attributes.

```
# Calculate expiration time (current time + 1 day in seconds)
EXPIRATION_TIME=$(date -d "+1 day" +%s)

# Add an item with TTL attribute
aws dynamodb put-item \
    --table-name TTLDemo \
    --item '{
        "ID": {"S": "item1"},
        "Data": {"S": "This item will expire in 1 day"},
        "ExpirationTime": {"N": "'$EXPIRATION_TIME'"}
    }'

# Add an item that expires in 1 hour
EXPIRATION_TIME_HOUR=$(date -d "+1 hour" +%s)
```

```
aws dynamodb put-item \
    --table-name TTLDemo \
    --item '{
        "ID": {"S": "item2"},
        "Data": {"S": "This item will expire in 1 hour"},
        "ExpirationTime": {"N": "'$EXPIRATION_TIME_HOUR'"}
    }'
```

Describe TTL settings.

```
# Describe TTL settings for a table
aws dynamodb describe-time-to-live \
    --table-name TTLDemo
```

- For API details, see the following topics in *AWS CLI Command Reference*.
  - AttachRolePolicy
  - CreateRole
  - CreateTable
  - DescribeTable
  - DescribeTimeToLive
  - PutItem
  - UpdateTimeToLive

For a complete list of AWS SDK developer guides and code examples, see Using DynamoDB with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Work with DynamoDB global tables and multi-Region replication with eventual consistency (MREC) using AWS Command Line Interface v2

The following code example shows how to manage DynamoDB global tables with multi-Region replication with eventual consistency (MREC).

- Create a table with multi-Region replication (MREC).
- Put and get items from replica tables.

- Remove replicas one-by-one.
- Clean up by deleting the table.

Bash

AWS CLI with Bash script

Create a table with multi-Region replication.

```
# Step 1: Create a new table (MusicTable) in US East (Ohio), with DynamoDB
 Streams enabled (NEW_AND_OLD_IMAGES)
aws dynamodb create-table \
    --table-name MusicTable \
    --attribute-definitions \
        AttributeName=Artist,AttributeType=S \
        AttributeName=SongTitle,AttributeType=S \
    --key-schema \
        AttributeName=Artist,KeyType=HASH \
        AttributeName=SongTitle,KeyType=RANGE \
    --billing-mode PAY_PER_REQUEST \
    --stream-specification StreamEnabled=true,StreamViewType=NEW_AND_OLD_IMAGES \
    --region us-east-2

# Step 2: Create an identical MusicTable table in US East (N. Virginia)
aws dynamodb update-table --table-name MusicTable --cli-input-json \
'{
  "ReplicaUpdates":
  [
    {
      "Create": {
        "RegionName": "us-east-1"
      }
    }
  ]
}' \
--region us-east-2

# Step 3: Create a table in Europe (Ireland)
aws dynamodb update-table --table-name MusicTable --cli-input-json \
'{
  "ReplicaUpdates":
  [
```

```
    {
      "Create": {
        "RegionName": "eu-west-1"
      }
    }
  ]
}' \
--region us-east-2
```

Describe the multi-Region table.

```
# Step 4: View the list of replicas created using describe-table
aws dynamodb describe-table \
    --table-name MusicTable \
    --region us-east-2 \
    --query 'Table.
{TableName:TableName,TableStatus:TableStatus,MultiRegionConsistency:MultiRegionConsistenc
{Region:RegionName,Status:ReplicaStatus}}'
```

Put items in a replica table.

```
# Step 5: To verify that replication is working, add a new item to the Music
 table in US East (Ohio)
aws dynamodb put-item \
    --table-name MusicTable \
    --item '{"Artist": {"S":"item_1"},"SongTitle": {"S":"Song Value 1"}}' \
    --region us-east-2
```

Get items from replica tables.

```
# Step 6: Wait for a few seconds, and then check to see whether the item has
 been
# successfully replicated to US East (N. Virginia) and Europe (Ireland)
aws dynamodb get-item \
    --table-name MusicTable \
    --key '{"Artist": {"S":"item_1"},"SongTitle": {"S":"Song Value 1"}}' \
    --region us-east-1

aws dynamodb get-item \
```

```
    --table-name MusicTable \
    --key '{"Artist": {"S":"item_1"},"SongTitle": {"S":"Song Value 1"}}' \
    --region eu-west-1
```

Remove replicas.

```
# Step 7: Delete the replica table in Europe (Ireland) Region
aws dynamodb update-table --table-name MusicTable --cli-input-json \
'{
  "ReplicaUpdates":
  [
    {
      "Delete": {
        "RegionName": "eu-west-1"
      }
    }
  ]
}' \
--region us-east-2

# Delete the replica table in US East (N. Virginia) Region
aws dynamodb update-table --table-name MusicTable --cli-input-json \
'{
  "ReplicaUpdates":
  [
    {
      "Delete": {
        "RegionName": "us-east-1"
      }
    }
  ]
}' \
--region us-east-2
```

Clean up by deleting the table.

```
# Clean up: Delete the primary table
aws dynamodb delete-table --table-name MusicTable --region us-east-2

echo "Global table demonstration complete."
```

- For API details, see the following topics in *AWS CLI Command Reference*.
  - CreateTable
  - DeleteTable
  - DescribeTable
  - GetItem
  - PutItem
  - UpdateTable

For a complete list of AWS SDK developer guides and code examples, see Using DynamoDB with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Work with DynamoDB resource tagging using AWS Command Line Interface v2

The following code example shows how to manage tags for DynamoDB resources.

- Create a table with tags.
- List tags for a resource.
- Add tags to a resource.
- Remove tags from a resource.
- Filter tables by tags.

Bash

AWS CLI with Bash script

Create a table with tags.

```
# Create a table with tags
aws dynamodb create-table \
    --table-name TaggedTable \
    --attribute-definitions \
        AttributeName=ID,AttributeType=S \
```

```
    --key-schema \
        AttributeName=ID,KeyType=HASH \
    --billing-mode PAY_PER_REQUEST \
    --tags \
        Key=Environment,Value=Production \
        Key=Project,Value=Analytics \
        Key=Owner,Value=DataTeam
```

List tags for a resource.

```
# Get the table ARN
TABLE_ARN=$(aws dynamodb describe-table \
    --table-name TaggedTable \
    --query "Table.TableArn" \
    --output text)

# List tags for the table
aws dynamodb list-tags-of-resource \
    --resource-arn $TABLE_ARN
```

Add tags to a resource.

```
# Add tags to an existing table
aws dynamodb tag-resource \
    --resource-arn $TABLE_ARN \
    --tags \
        Key=CostCenter,Value=12345 \
        Key=BackupSchedule,Value=Daily
```

Remove tags from a resource.

```
# Remove tags from a table
aws dynamodb untag-resource \
    --resource-arn $TABLE_ARN \
    --tag-keys Owner BackupSchedule
```

Filter tables by tags.

```
# Create another table with different tags
aws dynamodb create-table \
    --table-name AnotherTaggedTable \
    --attribute-definitions \
        AttributeName=ID,AttributeType=S \
    --key-schema \
        AttributeName=ID,KeyType=HASH \
    --billing-mode PAY_PER_REQUEST \
    --tags \
        Key=Environment,Value=Development \
        Key=Project,Value=Testing

# Wait for table to become active
aws dynamodb wait table-exists --table-name AnotherTaggedTable

# List all tables
echo "All tables:"
aws dynamodb list-tables

# Get ARNs for all tables
echo -e "\nFiltering tables by Environment=Production tag:"
TABLE_ARNS=$(aws dynamodb list-tables --query "TableNames[*]" --output text |
 xargs -I {} aws dynamodb describe-table --table-name {} --query "Table.TableArn"
 --output text)

# Find tables with specific tag
for ARN in $TABLE_ARNS; do
    TABLE_NAME=$(echo $ARN | awk -F/ '{print $2}')
    TAGS=$(aws dynamodb list-tags-of-resource --resource-arn $ARN --query "Tags[?
Key=='Environment' && Value=='Production']" --output text)
    if [ ! -z "$TAGS" ]; then
        echo "Table with Production tag: $TABLE_NAME"
    fi
done
```

- For API details, see the following topics in *AWS CLI Command Reference*.
  - CreateTable
  - ListTagsOfResource
  - TagResource
  - UntagResource

For a complete list of AWS SDK developer guides and code examples, see Using DynamoDB with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Work with DynamoDB table encryption using AWS Command Line Interface v2

The following code example shows how to manage encryption options for DynamoDB tables.

- Create a table with default encryption.
- Create a table with a customer managed CMK.
- Update table encryption settings.
- Describe table encryption.

Bash

AWS CLI with Bash script

Create a table with default encryption.

```
# Create a table with default encryption (AWS owned key)
aws dynamodb create-table \
    --table-name CustomerData \
    --attribute-definitions \
        AttributeName=CustomerID,AttributeType=S \
    --key-schema \
        AttributeName=CustomerID,KeyType=HASH \
    --billing-mode PAY_PER_REQUEST \
    --sse-specification Enabled=true,SSEType=KMS
```

Create a table with a customer managed CMK.

```
# Step 1: Create a customer managed key in KMS
aws kms create-key \
    --description "Key for DynamoDB table encryption" \
    --key-usage ENCRYPT_DECRYPT \
    --customer-master-key-spec SYMMETRIC_DEFAULT

# Store the key ID for later use
```

```
KEY_ID=$(aws kms list-keys --query "Keys[?contains(KeyArn, 'Key for
 DynamoDB')].KeyId" --output text)

# Step 2: Create a table with the customer managed key
aws dynamodb create-table \
    --table-name SensitiveData \
    --attribute-definitions \
        AttributeName=RecordID,AttributeType=S \
    --key-schema \
        AttributeName=RecordID,KeyType=HASH \
    --billing-mode PAY_PER_REQUEST \
    --sse-specification Enabled=true,SSEType=KMS,KMSMasterKeyId=$KEY_ID
```

Update table encryption.

```
# Update a table to use a different KMS key
aws dynamodb update-table \
    --table-name CustomerData \
    --sse-specification Enabled=true,SSEType=KMS,KMSMasterKeyId=$KEY_ID
```

Describe table encryption.

```
# Describe the table to see encryption settings
aws dynamodb describe-table \
    --table-name CustomerData \
    --query "Table.SSEDescription"
```

- For API details, see the following topics in *AWS CLI Command Reference*.
  - CreateKey
  - CreateTable
  - DescribeTable
  - UpdateTable

For a complete list of AWS SDK developer guides and code examples, see Using DynamoDB with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Serverless examples for DynamoDB

The following code examples show how to use DynamoDB with AWS SDKs.

Examples

- Invoke a Lambda function from a DynamoDB trigger
- Reporting batch item failures for Lambda functions with a DynamoDB trigger

Invoke a Lambda function from a DynamoDB trigger

The following code examples show how to implement a Lambda function that receives an event triggered by receiving records from a DynamoDB stream. The function retrieves the DynamoDB payload and logs the record contents.

.NET

SDK for .NET

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming a DynamoDB event with Lambda using .NET.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using System.Text.Json;
using System.Text;
using Amazon.Lambda.Core;
using Amazon.Lambda.DynamoDBEvents;

// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace AWSLambda_DDB;
```

```
public class Function
{
    public void FunctionHandler(DynamoDBEvent dynamoEvent, ILambdaContext
 context)
    {
        context.Logger.LogInformation($"Beginning to process
 {dynamoEvent.Records.Count} records...");

        foreach (var record in dynamoEvent.Records)
        {
            context.Logger.LogInformation($"Event ID: {record.EventID}");
            context.Logger.LogInformation($"Event Name: {record.EventName}");

            context.Logger.LogInformation(JsonSerializer.Serialize(record));
        }

        context.Logger.LogInformation("Stream processing complete.");
    }
}
```

Go

SDK for Go V2

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming a DynamoDB event with Lambda using Go.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package main

import (
 "context"
 "github.com/aws/aws-lambda-go/lambda"
 "github.com/aws/aws-lambda-go/events"
```

```
 "fmt"
)

func HandleRequest(ctx context.Context, event events.DynamoDBEvent) (*string,
 error) {
 if len(event.Records) == 0 {
  return nil, fmt.Errorf("received empty event")
 }

 for _, record := range event.Records {
   LogDynamoDBRecord(record)
 }

 message := fmt.Sprintf("Records processed: %d", len(event.Records))
 return &message, nil
}

func main() {
 lambda.Start(HandleRequest)
}

func LogDynamoDBRecord(record events.DynamoDBEventRecord){
 fmt.Println(record.EventID)
 fmt.Println(record.EventName)
 fmt.Printf("%+v\n", record.Change)
}
```

Java

SDK for Java 2.x

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming a DynamoDB event with Lambda using Java.

```
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.DynamodbEvent;
```

```
import
 com.amazonaws.services.lambda.runtime.events.DynamodbEvent.DynamodbStreamRecord;
import com.google.gson.Gson;
import com.google.gson.GsonBuilder;

public class example implements RequestHandler<DynamodbEvent, Void> {

    private static final Gson GSON = new
 GsonBuilder().setPrettyPrinting().create();

    @Override
    public Void handleRequest(DynamodbEvent event, Context context) {
        System.out.println(GSON.toJson(event));
        event.getRecords().forEach(this::logDynamoDBRecord);
        return null;
    }

    private void logDynamoDBRecord(DynamodbStreamRecord record) {
        System.out.println(record.getEventID());
        System.out.println(record.getEventName());
        System.out.println("DynamoDB Record: " +
 GSON.toJson(record.getDynamodb()));
    }
}
```

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming a DynamoDB event with Lambda using JavaScript.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
exports.handler = async (event, context) => {
    console.log(JSON.stringify(event, null, 2));
    event.Records.forEach(record => {
```

```
        logDynamoDBRecord(record);
    });
};

const logDynamoDBRecord = (record) => {
    console.log(record.eventID);
    console.log(record.eventName);
    console.log(`DynamoDB Record: ${JSON.stringify(record.dynamodb)}`);
};
```

Consuming a DynamoDB event with Lambda using TypeScript.

```
export const handler = async (event, context) => {
    console.log(JSON.stringify(event, null, 2));
    event.Records.forEach(record => {
        logDynamoDBRecord(record);
    });
}
const logDynamoDBRecord = (record) => {
    console.log(record.eventID);
    console.log(record.eventName);
    console.log(`DynamoDB Record: ${JSON.stringify(record.dynamodb)}`);
};
```

PHP

SDK for PHP

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming a DynamoDB event with Lambda using PHP.

```
<?php

# using bref/bref and bref/logger for simplicity
```

```
use Bref\Context\Context;
use Bref\Event\DynamoDb\DynamoDbEvent;
use Bref\Event\DynamoDb\DynamoDbHandler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';

class Handler extends DynamoDbHandler
{
    private StderrLogger $logger;

    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }

    /**
     * @throws JsonException
     * @throws \Bref\Event\InvalidLambdaEvent
     */
    public function handleDynamoDb(DynamoDbEvent $event, Context $context): void
    {
        $this->logger->info("Processing DynamoDb table items");
        $records = $event->getRecords();

        foreach ($records as $record) {
            $eventName = $record->getEventName();
            $keys = $record->getKeys();
            $old = $record->getOldImage();
            $new = $record->getNewImage();

            $this->logger->info("Event Name:".$eventName."\n");
            $this->logger->info("Keys:". json_encode($keys)."\n");
            $this->logger->info("Old Image:". json_encode($old)."\n");
            $this->logger->info("New Image:". json_encode($new));

            // TODO: Do interesting work based on the new data

            // Any exception thrown will be logged and the invocation will be
 marked as failed
        }

        $totalRecords = count($records);
        $this->logger->info("Successfully processed $totalRecords items");
```

```
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming a DynamoDB event with Lambda using Python.

```
import json

def lambda_handler(event, context):
    print(json.dumps(event, indent=2))

    for record in event['Records']:
        log_dynamodb_record(record)

def log_dynamodb_record(record):
    print(record['eventID'])
    print(record['eventName'])
    print(f"DynamoDB Record: {json.dumps(record['dynamodb'])}")
```

Ruby

SDK for Ruby

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming a DynamoDB event with Lambda using Ruby.

```
def lambda_handler(event:, context:)
    return 'received empty event' if event['Records'].empty?

    event['Records'].each do |record|
      log_dynamodb_record(record)
    end

    "Records processed: #{event['Records'].length}"
  end

  def log_dynamodb_record(record)
    puts record['eventID']
    puts record['eventName']
    puts "DynamoDB Record: #{JSON.generate(record['dynamodb'])}"
  end
```

Rust

SDK for Rust

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming a DynamoDB event with Lambda using Rust.

```
use lambda_runtime::{service_fn, tracing, Error, LambdaEvent};
use aws_lambda_events::{
    event::dynamodb::{Event, EventRecord},
   };


// Built with the following dependencies:
//lambda_runtime = "0.11.1"
//serde_json = "1.0"
//tokio = { version = "1", features = ["macros"] }
//tracing = { version = "0.1", features = ["log"] }
//tracing-subscriber = { version = "0.3", default-features = false, features =
 ["fmt"] }
//aws_lambda_events = "0.15.0"

async fn function_handler(event: LambdaEvent<Event>) ->Result<(), Error> {

    let records = &event.payload.records;
    tracing::info!("event payload: {:?}",records);
    if records.is_empty() {
        tracing::info!("No records found. Exiting.");
        return Ok(());
    }

    for record in records{
        log_dynamo_dbrecord(record);
    }

    tracing::info!("Dynamo db records processed");

    // Prepare the response
    Ok(())

}

fn log_dynamo_dbrecord(record: &EventRecord)-> Result<(), Error>{
    tracing::info!("EventId: {}", record.event_id);
    tracing::info!("EventName: {}", record.event_name);
    tracing::info!("DynamoDB Record: {:?}", record.change );
    Ok(())
```

```
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
    .with_max_level(tracing::Level::INFO)
    .with_target(false)
    .without_time()
    .init();

    let func = service_fn(function_handler);
    lambda_runtime::run(func).await?;
    Ok(())

}
```

For a complete list of AWS SDK developer guides and code examples, see Using DynamoDB with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Reporting batch item failures for Lambda functions with a DynamoDB trigger

The following code examples show how to implement partial batch response for Lambda functions that receive events from a DynamoDB stream. The function reports the batch item failures in the response, signaling to Lambda to retry those messages later.

.NET

SDK for .NET

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Reporting DynamoDB batch item failures with Lambda using .NET.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using System.Text.Json;
using System.Text;
using Amazon.Lambda.Core;
using Amazon.Lambda.DynamoDBEvents;

// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace AWSLambda_DDB;

public class Function
{
    public StreamsEventResponse FunctionHandler(DynamoDBEvent dynamoEvent,
 ILambdaContext context)

    {
        context.Logger.LogInformation($"Beginning to process
 {dynamoEvent.Records.Count} records...");
        List<StreamsEventResponse.BatchItemFailure> batchItemFailures = new
 List<StreamsEventResponse.BatchItemFailure>();
        StreamsEventResponse streamsEventResponse = new StreamsEventResponse();

        foreach (var record in dynamoEvent.Records)
        {
            try
            {
                var sequenceNumber = record.Dynamodb.SequenceNumber;
                context.Logger.LogInformation(sequenceNumber);
            }
            catch (Exception ex)
            {
                context.Logger.LogError(ex.Message);
                batchItemFailures.Add(new StreamsEventResponse.BatchItemFailure()
 { ItemIdentifier = record.Dynamodb.SequenceNumber });
            }
        }

        if (batchItemFailures.Count > 0)
        {
```

```
            streamsEventResponse.BatchItemFailures = batchItemFailures;
        }

        context.Logger.LogInformation("Stream processing complete.");
        return streamsEventResponse;
    }
}
```

Go

SDK for Go V2

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Reporting DynamoDB batch item failures with Lambda using Go.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package main

import (
 "context"
 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)

type BatchItemFailure struct {
 ItemIdentifier string `json:"ItemIdentifier"`
}

type BatchResult struct {
 BatchItemFailures []BatchItemFailure `json:"BatchItemFailures"`
}

func HandleRequest(ctx context.Context, event events.DynamoDBEvent)
 (*BatchResult, error) {
 var batchItemFailures []BatchItemFailure
 curRecordSequenceNumber := ""
```

```
 for _, record := range event.Records {
  // Process your record
  curRecordSequenceNumber = record.Change.SequenceNumber
 }

 if curRecordSequenceNumber != "" {
  batchItemFailures = append(batchItemFailures, BatchItemFailure{ItemIdentifier:
 curRecordSequenceNumber})
 }

 batchResult := BatchResult{
  BatchItemFailures: batchItemFailures,
 }

 return &batchResult, nil
}

func main() {
 lambda.Start(HandleRequest)
}
```

Java

SDK for Java 2.x

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Reporting DynamoDB batch item failures with Lambda using Java.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.DynamodbEvent;
import com.amazonaws.services.lambda.runtime.events.StreamsEventResponse;
import com.amazonaws.services.lambda.runtime.events.models.dynamodb.StreamRecord;
```

```
import java.util.ArrayList;
import java.util.List;

public class ProcessDynamodbRecords implements RequestHandler<DynamodbEvent,
 StreamsEventResponse> {

    @Override
    public StreamsEventResponse handleRequest(DynamodbEvent input, Context
 context) {

        List<StreamsEventResponse.BatchItemFailure> batchItemFailures = new
 ArrayList<>();
        String curRecordSequenceNumber = "";

        for (DynamodbEvent.DynamodbStreamRecord dynamodbStreamRecord :
 input.getRecords()) {
          try {
                //Process your record
                StreamRecord dynamodbRecord = dynamodbStreamRecord.getDynamodb();
                curRecordSequenceNumber = dynamodbRecord.getSequenceNumber();

            } catch (Exception e) {
                /* Since we are working with streams, we can return the failed
 item immediately.
                   Lambda will immediately begin to retry processing from this
 failed item onwards. */
                batchItemFailures.add(new
 StreamsEventResponse.BatchItemFailure(curRecordSequenceNumber));
                return new StreamsEventResponse(batchItemFailures);
            }
        }

       return new StreamsEventResponse();
    }
}
```

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Reporting DynamoDB batch item failures with Lambda using JavaScript.

```
export const handler = async (event) => {
  const records = event.Records;
  let curRecordSequenceNumber = "";

  for (const record of records) {
    try {
      // Process your record
      curRecordSequenceNumber = record.dynamodb.SequenceNumber;
    } catch (e) {
      // Return failed record's sequence number
      return { batchItemFailures: [{ itemIdentifier:
 curRecordSequenceNumber }] };
    }
  }

  return { batchItemFailures: [] };
};
```

Reporting DynamoDB batch item failures with Lambda using TypeScript.

```
import {
  DynamoDBBatchResponse,
  DynamoDBBatchItemFailure,
  DynamoDBStreamEvent,
} from "aws-lambda";

export const handler = async (
  event: DynamoDBStreamEvent
): Promise<DynamoDBBatchResponse> => {
```

```
  const batchItemFailures: DynamoDBBatchItemFailure[] = [];
  let curRecordSequenceNumber;

  for (const record of event.Records) {
    curRecordSequenceNumber = record.dynamodb?.SequenceNumber;

    if (curRecordSequenceNumber) {
      batchItemFailures.push({
        itemIdentifier: curRecordSequenceNumber,
      });
    }
  }

  return { batchItemFailures: batchItemFailures };
};
```

PHP

SDK for PHP

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Reporting DynamoDB batch item failures with Lambda using PHP.

```
<?php

# using bref/bref and bref/logger for simplicity

use Bref\Context\Context;
use Bref\Event\DynamoDb\DynamoDbEvent;
use Bref\Event\Handler as StdHandler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';

class Handler implements StdHandler
{
```

```
    private StderrLogger $logger;
    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }

    /**
     * @throws JsonException
     * @throws \Bref\Event\InvalidLambdaEvent
     */
    public function handle(mixed $event, Context $context): array
    {
        $dynamoDbEvent = new DynamoDbEvent($event);
        $this->logger->info("Processing records");

        $records = $dynamoDbEvent->getRecords();
        $failedRecords = [];
        foreach ($records as $record) {
            try {
                $data = $record->getData();
                $this->logger->info(json_encode($data));
                // TODO: Do interesting work based on the new data
            } catch (Exception $e) {
                $this->logger->error($e->getMessage());
                // failed processing the record
                $failedRecords[] = $record->getSequenceNumber();
            }
        }
        $totalRecords = count($records);
        $this->logger->info("Successfully processed $totalRecords records");

        // change format for the response
        $failures = array_map(
            fn(string $sequenceNumber) => ['itemIdentifier' => $sequenceNumber],
            $failedRecords
        );

        return [
            'batchItemFailures' => $failures
        ];
    }
}

$logger = new StderrLogger();
```

```
return new Handler($logger);
```

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Reporting DynamoDB batch item failures with Lambda using Python.

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
def handler(event, context):
    records = event.get("Records")
    curRecordSequenceNumber = ""

    for record in records:
        try:
            # Process your record
            curRecordSequenceNumber = record["dynamodb"]["SequenceNumber"]
        except Exception as e:
            # Return failed record's sequence number
            return {"batchItemFailures":[{"itemIdentifier":
 curRecordSequenceNumber}]}

    return {"batchItemFailures":[]}
```

Ruby

SDK for Ruby

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Reporting DynamoDB batch item failures with Lambda using Ruby.

```
def lambda_handler(event:, context:)
    records = event["Records"]
    cur_record_sequence_number = ""

    records.each do |record|
      begin
        # Process your record
        cur_record_sequence_number = record["dynamodb"]["SequenceNumber"]
      rescue StandardError => e
        # Return failed record's sequence number
        return {"batchItemFailures" => [{"itemIdentifier" =>
 cur_record_sequence_number}]}
      end
    end

    {"batchItemFailures" => []}
  end
```

Rust

SDK for Rust

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Reporting DynamoDB batch item failures with Lambda using Rust.

```
use aws_lambda_events::{
    event::dynamodb::{Event, EventRecord, StreamRecord},
    streams::{DynamoDbBatchItemFailure, DynamoDbEventResponse},
};
use lambda_runtime::{run, service_fn, Error, LambdaEvent};

/// Process the stream record
fn process_record(record: &EventRecord) -> Result<(), Error> {
    let stream_record: &StreamRecord = &record.change;

    // process your stream record here...
    tracing::info!("Data: {:?}", stream_record);

    Ok(())
}

/// Main Lambda handler here...
async fn function_handler(event: LambdaEvent<Event>) ->
 Result<DynamoDbEventResponse, Error> {
    let mut response = DynamoDbEventResponse {
        batch_item_failures: vec![],
    };

    let records = &event.payload.records;

    if records.is_empty() {
        tracing::info!("No records found. Exiting.");
        return Ok(response);
    }

    for record in records {
        tracing::info!("EventId: {}", record.event_id);

        // Couldn't find a sequence number
        if record.change.sequence_number.is_none() {
            response.batch_item_failures.push(DynamoDbBatchItemFailure {
                item_identifier: Some("".to_string()),
            });
            return Ok(response);
        }

        // Process your record here...
```

```
        if process_record(record).is_err() {
            response.batch_item_failures.push(DynamoDbBatchItemFailure {
                item_identifier: record.change.sequence_number.clone(),
            });
            /* Since we are working with streams, we can return the failed item
 immediately.
            Lambda will immediately begin to retry processing from this failed
 item onwards. */
            return Ok(response);
        }
    }

    tracing::info!("Successfully processed {} record(s)", records.len());

    Ok(response)
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
        .with_max_level(tracing::Level::INFO)
        // disable printing the name of the module in every log line.
        .with_target(false)
        // disabling time is handy because CloudWatch will add the ingestion
 time.
        .without_time()
        .init();

    run(service_fn(function_handler)).await
}
```

For a complete list of AWS SDK developer guides and code examples, see Using DynamoDB with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

AWS community contributions for DynamoDB

AWS community contributions are examples that were created and are maintained by multiple teams across AWS. To provide feedback, use the mechanism provided in the linked repositories.

Examples

- Build and test a serverless application

Build and test a serverless application

The following code examples show how to build and test a serverless application using API Gateway with Lambda and DynamoDB

.NET

SDK for .NET

Shows how to build and test a serverless application that consists of an API Gateway with Lambda and DynamoDB using the .NET SDK.

For complete source code and instructions on how to set up and run, see the full example on GitHub.

Services used in this example

- API Gateway
- DynamoDB
- Lambda

Go

SDK for Go V2

Shows how to build and test a serverless application that consists of an API Gateway with Lambda and DynamoDB using the Go SDK.

For complete source code and instructions on how to set up and run, see the full example on GitHub.

Services used in this example

- API Gateway
- DynamoDB
- Lambda

Java

SDK for Java 2.x

Shows how to build and test a serverless application that consists of an API Gateway with Lambda and DynamoDB using the Java SDK.

For complete source code and instructions on how to set up and run, see the full example on GitHub.

Services used in this example

- API Gateway
- DynamoDB
- Lambda

Rust

SDK for Rust

Shows how to build and test a serverless application that consists of an API Gateway with Lambda and DynamoDB using the Rust SDK.

For complete source code and instructions on how to set up and run, see the full example on GitHub.

Services used in this example

- API Gateway
- DynamoDB
- Lambda

For a complete list of AWS SDK developer guides and code examples, see Using DynamoDB with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Security and compliance in Amazon DynamoDB

Cloud security at AWS is the highest priority. As an AWS customer, you benefit from a data center and network architecture that is built to meet the requirements of the most security-sensitive organizations.

Security is a shared responsibility between AWS and you. The shared responsibility model describes this as security *of* the cloud and security *in* the cloud:

- **Security of the cloud** – AWS is responsible for protecting the infrastructure that runs AWS services in the AWS Cloud. AWS also provides you with services that you can use securely. The effectiveness of our security is regularly tested and verified by third-party auditors as part of the AWS compliance programs. To learn about the compliance programs that apply to DynamoDB, see AWS Services in Scope by Compliance Program.
- **Security in the cloud** – Your responsibility is determined by the AWS service that you use. You are also responsible for other factors including the sensitivity of your data, your organization's requirements, and applicable laws and regulations.

This documentation will help you understand how to apply the shared responsibility model when using DynamoDB. The following topics show you how to configure DynamoDB to meet your security and compliance objectives. You'll also learn how to use other AWS services that can help you to monitor and secure your DynamoDB resources.

Topics

- AWS managed policies for Amazon DynamoDB
- Using resource-based policies for DynamoDB
- Using attribute-based access control with DynamoDB
- Data protection in DynamoDB
- AWS Identity and Access Management (IAM) and DynamoDB
- Compliance validation by industry for DynamoDB
- Resilience and disaster recovery in Amazon DynamoDB
- Infrastructure security in Amazon DynamoDB
- AWS PrivateLink for DynamoDB
- Configuration and vulnerability analysis in Amazon DynamoDB

- Security best practices for Amazon DynamoDB

AWS managed policies for Amazon DynamoDB

DynamoDB uses AWS managed policies to define a set of permissions the service needs to perform specific actions. DynamoDB maintains and updates its AWS managed policies. You can't change the permissions in AWS managed policies. For more information about AWS managed policies, see AWS managed policies in the IAM User Guide.

DynamoDB may occasionally add additional permissions to an AWS managed policy to support new features. This type of update affects all identities (users, groups, and roles) where the policy is attached. An AWS managed policy is most likely to be updated when a new feature is launched or when new operations become available. DynamoDB will not remove permissions from an AWS managed policy, so policy updates won't break your existing permissions. For a full list of AWS managed policies, see AWS managed policies.

AWS managed policy: DynamoDBReplicationServiceRolePolicy

You can’t attach the `DynamoDBReplicationServiceRolePolicy` policy to your IAM entities. This policy is attached to a service-linked role that allows DynamoDB to perform actions on your behalf. For more information, see Using IAM with global tables.

This policy grants permissions that allow the service-linked role to perform data replication between global table replicas. It also grants administrative permissions to manage global table replicas on your behalf.

Permissions details

This policy grants permissions to do the following:

- `dynamodb` – Perform data replication and manage table replicas.
- `application-autoscaling` – Retrieve and manage table Auto Scaling settings
- `account` – Retrieve region status for evaluating replica accessibility.
- `iam` – To create the service-linked role for application Auto Scaling in the event that the service-linked role does not already exist.

The definition of this managed policy can be found here.

AWS managed policy: AmazonDynamoDBFullAccess_v2

The scoped-down AmazonDynamoDBFullAccess_v2 policy grants specific access privileges to users. You can attach the AmazonDynamoDBFullAccess_v2 policy to your IAM identities. This policy grants administrative access to Amazon DynamoDB resources and grants an IAM identity (such as a user, group, or role) access to the AWS services that DynamoDB is integrated with to use all of DynamoDB features. Using this policy allows access to all of DynamoDB features that are available in the AWS Management Console.

Permissions details

This policy includes the following permissions:

- Amazon DynamoDB
- DynamoDB Accelerator
- AWS KMS
- AWS Resource Groups Tagging
- Lambda
- Application Auto Scaling
- CloudWatch
- Amazon Kinesis
- Amazon EC2
- IAM

To review the policy in JSON format, see AmazonDynamoDBFullAccess_v2.

AWS managed policy: AmazonDynamoDBReadOnlyAccess

You can attach the AmazonDynamoDBReadOnlyAccess policy to your IAM identities.

This policy grants read-only access to Amazon DynamoDB.

Permissions details

This policy includes the following permissions:

- `Amazon DynamoDB` – Provides read-only access to Amazon DynamoDB.
- `Amazon DynamoDB Accelerator (DAX)` – Provides read-only access to Amazon DynamoDB Accelerator (DAX).
- `Application Auto Scaling` – Allows principals to view configurations from Application Auto Scaling. This is required so that users can view automatic scaling policies that are attached to a table.
- `CloudWatch` – Allows principals to view metric data and alarms configured in CloudWatch. This is required so users can view the billable table size and CloudWatch alarms that have been configured for a table.
- `AWS Data Pipeline` – Allows principals to view AWS Data Pipeline and associated objects.
- `Amazon EC2` – Allows principals to view Amazon EC2 VPCs, subnets, and security groups.
- `IAM` – Allows principals to view IAM roles.
- `AWS KMS` – Allows principals to view keys configured in AWS KMS. This is required so users can view AWS KMS keys that they create and manage in their account.
- `Amazon SNS` – Allows principals to list Amazon SNS topics and subscriptions by topic.
- `AWS Resource Groups` – Allows principals to view resource groups and their queries.
- `AWS Resource Groups Tagging` – Allows principals to list all the tagged or previously tagged resources in a Region.
- `Kinesis` – Allows principals to view Kinesis data streams descriptions.
- `Amazon CloudWatch Contributor Insights` – Allow principals to view time series data collected by Contributor Insights rules.

To review the policy in `JSON` format, see AmazonDynamoDBReadOnlyAccess.

DynamoDB updates to AWS managed policies

This table shows updates to the AWS access management policies for DynamoDB.

Change	Description	Date Changed
`AmazonDynamoDBFullAccess` – Deprecated	This policy has been replaced by a scoped-down policy named `AmazonDynamoDBFullAccess_v2` .	April 28, 2025

Change	Description	Date Changed
	The AmazonDynamoDBFullAccess policy is deprecated and is no longer recommended.	
AmazonDynamoDBReadOnlyAccess update to an existing policy	AmazonDynamoDBReadOnlyAccess added the permissions: dynamodb:GetAbacStatus and dynamodb:UpdateAbacStatus. These permissions allow you to view the ABAC status and enable ABAC for your AWS account in the current Region.	November 18, 2024
AmazonDynamoDBReadOnlyAccess update to an existing policy	AmazonDynamoDBReadOnlyAccess added the permission dynamodb:GetResourcePolicy. This permission provides access to read resource-based policies attached to DynamoDB resources.	March 20, 2024
DynamoDBReplicationServiceRolePolicy update to an existing policy	DynamoDBReplicationServiceRolePolicy added the permission dynamodb:GetResourcePolicy. This permission allows the service-linked role to read resource-based policies attached to DynamoDB resources.	December 15, 2023
DynamoDBReplicationServiceRolePolicy update to an existing policy	DynamoDBReplicationServiceRolePolicy added the permission account:ListRegions. This permission allows the service-linked role to evaluate replica accessibility	May 10, 2023

Change	Description	Date Changed
DynamoDBR eplicatio nServiceR olePolicy added to list of managed policies	Added information about the managed policy DynamoDBR eplicationServiceR olePolicy , which is used by the DynamoDB global tables service-l inked role.	May 10, 2023
DynamoDB global tables started tracking changes	DynamoDB global tables started tracking changes for its AWS managed policies.	May 10, 2023

Using resource-based policies for DynamoDB

DynamoDB supports resource-based policies for tables, indexes, and streams. Resource-based policies let you define access permissions by specifying who has access to each resource, and the actions they are allowed to perform on each resource.

You can attach a resource-based policy to DynamoDB resources, such as a table or a stream. In this policy, you specify permissions for Identity and Access Management (IAM) principals that can perform specific actions on these DynamoDB resources. For example, the policy attached to a table will contain permissions for access to the table and its indexes. As a result, resource-based policies can help you simplify access control for your DynamoDB tables, indexes, and streams, by defining permissions at the resource level. The maximum size of a policy you can attach to a DynamoDB resource is 20 KB.

A significant benefit of using resource-based policies is to simplify cross-account access control for providing cross-account access to IAM principals in different AWS accounts. For more information, see Resource-based policy for cross-account access.

Resource-based policies also support integrations with IAM Access Analyzer external access analyzer and Block Public Access (BPA) capabilities. IAM Access Analyzer reports cross-account access to external entities specified in resource-based policies. It also provides visibility to help you refine permissions and conform to the least privilege principle. BPA helps you prevent public access

to your DynamoDB tables, indexes, and streams, and is automatically enabled in the resource-based policies creation and modification workflows.

Topics

- Create a table with a resource-based policy
- Attach a policy to an DynamoDB existing table
- Attach a resource-based policy to a DynamoDB stream
- Remove a resource-based policy from a DynamoDB table
- Cross-account access with resource-based policies in DynamoDB
- Blocking public access with resource-based policies in DynamoDB
- DynamoDB API operations supported by resource-based policies
- Authorization with IAM identity-based policies and DynamoDB resource-based policies
- DynamoDB resource-based policy examples
- DynamoDB resource-based policy considerations
- DynamoDB resource-based policy best practices

Create a table with a resource-based policy

You can add a resource-based policy while you create a table by using the DynamoDB console, CreateTable API, AWS CLI, AWS SDK, or an CloudFormation template.

AWS CLI

The following example creates a table named *MusicCollection* using the `create-table` AWS CLI command. This command also includes the `resource-policy` parameter that adds a resource-based policy to the table. This policy allows the user *John* to perform the RestoreTableToPointInTime, GetItem, and PutItem API actions on the table.

Remember to replace the *italicized* text with your resource-specific information.

```
aws dynamodb create-table \
    --table-name MusicCollection \
    --attribute-definitions AttributeName=Artist,AttributeType=S
 AttributeName=SongTitle,AttributeType=S \
    --key-schema AttributeName=Artist,KeyType=HASH
 AttributeName=SongTitle,KeyType=RANGE \
```

```
    --provisioned-throughput ReadCapacityUnits=5,WriteCapacityUnits=5 \
    --resource-policy \
        "{
            \"Version\": \"2012-10-17\",
            \"Statement\": [
              {
                    \"Effect\": \"Allow\",
                    \"Principal\": {
                        \"AWS\": \"arn:aws:iam::123456789012:user/John\"
                    },
                    \"Action\": [
                        \"dynamodb:RestoreTableToPointInTime\",
                        \"dynamodb:GetItem\",
                        \"dynamodb:DescribeTable\"
                    ],
                    \"Resource\": \"arn:aws:dynamodb:us-
west-2:123456789012:table/MusicCollection\"
                }
            ]
        }"
```

AWS Management Console

1. Sign in to the AWS Management Console and open the DynamoDB console at https://console.aws.amazon.com/dynamodb/.
2. On the dashboard, choose **Create table**.
3. In **Table details**, enter the table name, partition key, and sort key details.
4. In **Table settings**, choose **Customize settings**.
5. (Optional) Specify your options for **Table class**, **Capacity calculator**, **Read/write capacity settings**, **Secondary indexes**, **Encryption at rest**, and **Deletion protection**.
6. In **Resource-based policy**, add a policy to define the access permissions for the table and its indexes. In this policy, you specify who has access to these resources, and the actions they are allowed to perform on each resource. To add a policy, do one of the following:

   - Type or paste a JSON policy document. For details about the IAM policy language, see Creating policies using the JSON editor in the *IAM User Guide*.

> **Tip**
>
> To see examples of resource-based policies in the Amazon DynamoDB Developer Guide, choose **Policy examples**.

- Choose **Add new statement** to add a new statement and enter the information in the provided fields. Repeat this step for as many statements as you would like to add.

> **Important**
>
> Make sure that you resolve any security warnings, errors, or suggestions before you save your policy.

The following IAM policy example allows the user *John* to perform the RestoreTableToPointInTime, GetItem, and PutItem API actions on the table *MusicCollection*.

Remember to replace the *italicized* text with your resource-specific information.

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "AWS": "arn:aws:iam::123456789012:user/username"
      },
      "Action": [
        "dynamodb:RestoreTableToPointInTime",
        "dynamodb:GetItem",
        "dynamodb:PutItem"
      ],
      "Resource": "arn:aws:dynamodb:us-
east-1:123456789012:table/MusicCollection"
    }
```

```
  ]
}
```

7. (Optional) Choose **Preview external access** in the lower-right corner to preview how your new policy affects public and cross-account access to your resource. Before you save your policy, you can check whether it introduces new IAM Access Analyzer findings or resolves existing findings. If you don't see an active analyzer, choose **Go to Access Analyzer** to create an account analyzer in IAM Access Analyzer. For more information, see Preview access.
8. Choose **Create table**.

AWS CloudFormation template

Using the AWS::DynamoDB::Table resource

The following CloudFormation template creates a table with a stream using the AWS::DynamoDB::Table resource. This template also includes resource-based policies that are attached to both the table and the stream.

```
{
    "AWSTemplateFormatVersion": "2010-09-09",
    "Resources": {
        "MusicCollectionTable": {
            "Type": "AWS::DynamoDB::Table",
            "Properties": {
                "AttributeDefinitions": [
                    {
                        "AttributeName": "Artist",
                        "AttributeType": "S"
                    }
                ],
                "KeySchema": [
                    {
                        "AttributeName": "Artist",
                        "KeyType": "HASH"
                    }
                ],
                "BillingMode": "PROVISIONED",
                "ProvisionedThroughput": {
                    "ReadCapacityUnits": 5,
                    "WriteCapacityUnits": 5
                },
                "StreamSpecification": {
```

```
                "StreamViewType": "OLD_IMAGE",
                "ResourcePolicy": {
                  "PolicyDocument": {
                    "Version": "2012-10-17",
                    "Statement": [
                      {
                          "Principal": {
                              "AWS": "arn:aws:iam::111122223333:user/John"
                          },
                          "Effect": "Allow",
                          "Action": [
                              "dynamodb:GetRecords",
                              "dynamodb:GetShardIterator",
                              "dynamodb:DescribeStream"
                          ],
                          "Resource": "*"
                      }
                    ]
                  }
                }
              },
              "TableName": "MusicCollection",
              "ResourcePolicy": {
                  "PolicyDocument": {
                      "Version": "2012-10-17",
                      "Statement": [
                          {
                              "Principal": {
                                  "AWS": [
                                      "arn:aws:iam::111122223333:user/John"
                                  ]
                              },
                              "Effect": "Allow",
                              "Action": "dynamodb:GetItem",
                              "Resource": "*"
                          }
                      ]
                  }
              }
          }

      }
  }
```

```
}
```

Using the AWS::DynamoDB::GlobalTable resource

The following CloudFormation template creates a table with the AWS::DynamoDB::GlobalTable resource and attaches a resource-based policy to the table and its stream.

```
{
    "AWSTemplateFormatVersion": "2010-09-09",
    "Resources": {
        "GlobalMusicCollection": {
            "Type": "AWS::DynamoDB::GlobalTable",
            "Properties": {
                "TableName": "MusicCollection",
                "AttributeDefinitions": [{
                    "AttributeName": "Artist",
                    "AttributeType": "S"
                }],
                "KeySchema": [{
                    "AttributeName": "Artist",
                    "KeyType": "HASH"
                }],
                "BillingMode": "PAY_PER_REQUEST",
                "StreamSpecification": {
                    "StreamViewType": "NEW_AND_OLD_IMAGES"
                },
                "Replicas": [
                    {
                        "Region": "us-east-1",
                        "ResourcePolicy": {
                            "PolicyDocument": {
                                "Version": "2012-10-17",
                                "Statement": [{
                                    "Principal": {
                                        "AWS": [
                                            "arn:aws:iam::111122223333:user/John"
                                        ]
                                    },
                                    "Effect": "Allow",
                                    "Action": "dynamodb:GetItem",
                                    "Resource": "*"
                                }]
                            }
```

```
                        },
                        "ReplicaStreamSpecification": {
                            "ResourcePolicy": {
                                "PolicyDocument": {
                                    "Version": "2012-10-17",
                                    "Statement": [{
                                        "Principal": {
                                            "AWS":
 "arn:aws:iam::111122223333:user/John"
                                        },
                                        "Effect": "Allow",
                                        "Action": [
                                            "dynamodb:GetRecords",
                                            "dynamodb:GetShardIterator",
                                            "dynamodb:DescribeStream"
                                        ],
                                        "Resource": "*"
                                    }]
                                }
                            }
                        }
                    }
                ]
            }
        }
    }
}
```

Attach a policy to an DynamoDB existing table

You can attach a resource-based policy to an existing table or modify an existing policy by using the DynamoDB console, PutResourcePolicy API, the AWS CLI, AWS SDK, or an CloudFormation template.

AWS CLI example to attach a new policy

The following IAM policy example uses the `put-resource-policy` AWS CLI command to attach a resource-based policy to an existing table. This example allows the user *John* to perform the GetItem, PutItem, UpdateItem, and UpdateTable API actions on an existing table named *MusicCollection*.

Remember to replace the *italicized* text with your resource-specific information.

```
aws dynamodb put-resource-policy \
    --resource-arn arn:aws:dynamodb:us-west-2:123456789012:table/MusicCollection \
    --policy \
        "{
            \"Version\": \"2012-10-17\",
            \"Statement\": [
              {
                    \"Effect\": \"Allow\",
                    \"Principal\": {
                        \"AWS\": \"arn:aws:iam::111122223333:user/John\"
                    },
                    \"Action\": [
                        \"dynamodb:GetItem\",
                        \"dynamodb:PutItem\",
                        \"dynamodb:UpdateItem\",
                        \"dynamodb:UpdateTable\"
                    ],
                    \"Resource\": \"arn:aws:dynamodb:us-
west-2:123456789012:table/MusicCollection\"
                }
            ]
        }"
```

AWS CLI example to conditionally update an existing policy

To conditionally update an existing resource-based policy of a table, you can use the optional `expected-revision-id` parameter. The following example will only update the policy if it exists in DynamoDB and its current revision ID matches the provided `expected-revision-id` parameter.

```
aws dynamodb put-resource-policy \
    --resource-arn arn:aws:dynamodb:us-west-2:123456789012:table/MusicCollection \
    --expected-revision-id 1709841168699 \
    --policy \
        "{
            \"Version\": \"2012-10-17\",
            \"Statement\": [
              {
                    \"Effect\": \"Allow\",
                    \"Principal\": {
                        \"AWS\": \"arn:aws:iam::111122223333:user/John\"
                    },
```

```
                    \"Action\": [
                        \"dynamodb:GetItem\",
                        \"dynamodb:UpdateItem\",
                        \"dynamodb:UpdateTable\"
                    ],
                    \"Resource\": \"arn:aws:dynamodb:us-
west-2:123456789012:table/MusicCollection\"
                }
            ]
        }"
```

AWS Management Console

1. Sign in to the AWS Management Console and open the DynamoDB console at https://console.aws.amazon.com/dynamodb/.
2. From the dashboard, choose an existing table.
3. Navigate to the **Permissions** tab, and choose **Create table policy**.
4. In the resource-based policy editor, add the policy you would like to attach and choose **Create policy**.

   The following IAM policy example allows the user *John* to perform the GetItem, PutItem, UpdateItem, and UpdateTable API actions on an existing table named *MusicCollection*.

   Remember to replace the *italicized* text with your resource-specific information.

   JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "AWS": "arn:aws:iam::111122223333:user/username"
      },
      "Action": [
        "dynamodb:GetItem",
        "dynamodb:PutItem",
        "dynamodb:UpdateItem",
        "dynamodb:UpdateTable"
```

```
      ],
      "Resource": "arn:aws:dynamodb:us-
east-1:123456789012:table/MusicCollection"
    }
  ]
}
```

AWS SDK for Java 2.x

The following IAM policy example uses the `putResourcePolicy` method to attach a resource-based policy to an existing table. This policy allows a user to perform the GetItem API action on an existing table.

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.PutResourcePolicyRequest;

/**
 * Before running this Java V2 code example, set up your development
 * environment, including your credentials.
 *
 * For more information, see the following documentation topic:
 *
 * Get started with the AWS SDK for Java 2.x
 */
public class PutResourcePolicy {

    public static void main(String[] args) {
        final String usage = """

                Usage:
                    <tableArn> <allowedAWSPrincipal>

                Where:
                    tableArn - The Amazon DynamoDB table ARN to attach the policy to.
 For example, arn:aws:dynamodb:us-west-2:123456789012:table/MusicCollection.
                    allowedAWSPrincipal - Allowed AWS principal ARN that the example
 policy will give access to. For example, arn:aws:iam::123456789012:user/John.
                """;

        if (args.length != 2) {
```

```
            System.out.println(usage);
            System.exit(1);
        }

        String tableArn = args[0];
        String allowedAWSPrincipal = args[1];
        System.out.println("Attaching a resource-based policy to the Amazon DynamoDB
table with ARN " +
                tableArn);
        Region region = Region.US_WEST_2;
        DynamoDbClient ddb = DynamoDbClient.builder()
                .region(region)
                .build();

        String result = putResourcePolicy(ddb, tableArn, allowedAWSPrincipal);
        System.out.println("Revision ID for the attached policy is " + result);
        ddb.close();
    }

    public static String putResourcePolicy(DynamoDbClient ddb, String tableArn, String
allowedAWSPrincipal) {
        String policy = generatePolicy(tableArn, allowedAWSPrincipal);
        PutResourcePolicyRequest request = PutResourcePolicyRequest.builder()
                .policy(policy)
                .resourceArn(tableArn)
                .build();

        try {
            return ddb.putResourcePolicy(request).revisionId();
        } catch (DynamoDbException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }

        return "";
    }

    private static String generatePolicy(String tableArn, String allowedAWSPrincipal) {
        return "{\n" +
                "    \"Version\": \"2012-10-17\",\n" +,
                "    \"Statement\": [\n" +
                "        {\n" +
                "            \"Effect\": \"Allow\",\n" +
```

```
                "            \"Principal\": {\"AWS\":\"" + allowedAWSPrincipal + "\"},
\n" +
                "            \"Action\": [\n" +
                "                \"dynamodb:GetItem\"\n" +
                "            ],\n" +
                "            \"Resource\": \"" + tableArn + "\"\n" +
                "        }\n" +
                "    ]\n" +
                "}";
    }
}
```

Attach a resource-based policy to a DynamoDB stream

You can attach a resource-based policy to an existing table's stream or modify an existing policy by using the DynamoDB console, PutResourcePolicy API, the AWS CLI, AWS SDK, or an CloudFormation template.

> **Note**
>
> You can't attach a policy to a stream while creating it using the CreateTable or UpdateTable APIs. However, you can modify or delete a policy after a table is deleted. You can also modify or delete the policy of a disabled stream.

AWS CLI

The following IAM policy example uses the `put-resource-policy` AWS CLI command to attach a resource-based policy to the stream of a table named *`MusicCollection`*. This example allows the user *`John`* to perform the GetRecords, GetShardIterator, and DescribeStream API actions on the stream.

Remember to replace the *`italicized`* text with your resource-specific information.

```
aws dynamodb put-resource-policy \
    --resource-arn arn:aws:dynamodb:us-west-2:123456789012:table/MusicCollection/
stream/2024-02-12T18:57:26.492 \
    --policy \
        "{
            \"Version\": \"2012-10-17\",
            \"Statement\": [
```

```
            {
                  \"Effect\": \"Allow\",
                  \"Principal\": {
                      \"AWS\": \"arn:aws:iam::111122223333:user/John\"
                  },
                  \"Action\": [
                      \"dynamodb:GetRecords\",
                      \"dynamodb:GetShardIterator\",
                      \"dynamodb:DescribeStream\"
                  ],
                  \"Resource\": \"arn:aws:dynamodb:us-
west-2:123456789012:table/MusicCollection/stream/2024-02-12T18:57:26.492\"
                }
            ]
        }"
```

AWS Management Console

1. Sign in to the AWS Management Console and open the DynamoDB console at https://console.aws.amazon.com/dynamodb/.
2. On the DynamoDB console dashboard, choose **Tables** and then select an existing table.

   Make sure the table you select has streams turned on. For information about turning on streams for a table, see Enabling a stream.
3. Choose the **Permissions** tab.
4. In **Resource-based policy for active stream**, choose **Create stream policy**.
5. In the **Resource-based policy** editor, add a policy to define the access permissions for the stream. In this policy, you specify who has access to the stream and the actions they are allowed to perform on the stream. To add a policy, do one of the following:

   - Type or paste a JSON policy document. For details about the IAM policy language, see Creating policies using the JSON editor in the *IAM User Guide*.

     > **Tip**
     >
     > To see examples of resource-based policies in the Amazon DynamoDB Developer Guide, choose **Policy examples**.

   - Choose **Add new statement** to add a new statement and enter the information in the provided fields. Repeat this step for as many statements as you would like to add.

> **Important**
>
> Make sure that you resolve any security warnings, errors, or suggestions before you save your policy.

6. (Optional) Choose **Preview external access** in the lower-right corner to preview how your new policy affects public and cross-account access to your resource. Before you save your policy, you can check whether it introduces new IAM Access Analyzer findings or resolves existing findings. If you don't see an active analyzer, choose **Go to Access Analyzer** to create an account analyzer in IAM Access Analyzer. For more information, see Preview access.
7. Choose **Create policy**.

The following IAM policy example attaches a resource-based policy to the stream of a table named *MusicCollection*. This example allows the user *John* to perform the GetRecords, GetShardIterator, and DescribeStream API actions on the stream.

Remember to replace the *italicized* text with your resource-specific information.

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "AWS": "arn:aws:iam::111122223333:user/username"
      },
      "Action": [
        "dynamodb:GetRecords",
        "dynamodb:GetShardIterator",
        "dynamodb:DescribeStream"
      ],
      "Resource": [
        "arn:aws:dynamodb:us-east-1:123456789012:table/MusicCollection/
stream/2024-02-12T18:57:26.492"
      ]
    }
```

```
  ]
}
```

Remove a resource-based policy from a DynamoDB table

You can delete a resource-based policy from an existing table by using the DynamoDB console, DeleteResourcePolicy API, the AWS CLI, AWS SDK, or an CloudFormation template.

AWS CLI

The following example uses the `delete-resource-policy` AWS CLI command to remove a resource-based policy from a table named *`MusicCollection`*.

Remember to replace the *`italicized`* text with your resource-specific information.

```
aws dynamodb delete-resource-policy \
    --resource-arn arn:aws:dynamodb:us-west-2:123456789012:table/MusicCollection
```

AWS Management Console

1. Sign in to the AWS Management Console and open the DynamoDB console at https://console.aws.amazon.com/dynamodb/.
2. On the DynamoDB console dashboard, choose **Tables** and then select an existing table.
3. Choose **Permissions**.
4. From the **Manage policy** dropdown, choose **Delete policy**.
5. In the **Delete resource-based policy for table** dialog box, type **`confirm`** to confirm the delete action.
6. Choose **Delete**.

Cross-account access with resource-based policies in DynamoDB

Using a resource-based policy, you can provide cross-account access to resources available in different AWS accounts. All cross-account access allowed by the resource-based policies will be reported through IAM Access Analyzer external access findings if you have an analyzer in the same AWS Region as the resource. IAM Access Analyzer runs policy checks to validate your policy against IAM policy grammar and best practices. These checks generate findings and provide actionable recommendations to help you author policies that are functional and conform to security best

practices. You can view the active findings from IAM Access Analyzer in the **Permissions** tab of the DynamoDB console.

For information about validating policies by using IAM Access Analyzer, see IAM Access Analyzer policy validation in the *IAM User Guide*. To view a list of the warnings, errors, and suggestions that are returned by IAM Access Analyzer, see IAM Access Analyzer policy check reference.

To grant GetItem permission to a user A in account A for accessing a table B in account B, perform the following steps:

1. Attach a resource-based policy to table B that grants permission to user A for performing the `GetItem` action.
2. Attach an identity-based policy to user A that grants it permission to perform the `GetItem` action on table B.

Using the **Preview external access** option available in DynamoDB console, you can preview how your new policy affects public and cross-account access to your resource. Before you save your policy, you can check whether it introduces new IAM Access Analyzer findings or resolves existing findings. If you don't see an active analyzer, choose **Go to Access Analyzer** to create an account analyzer in IAM Access Analyzer. For more information, see Preview access.

The table name parameter in the DynamoDB data plane and control plane APIs accept complete Amazon Resource Name (ARN) of the table to support cross-account operations. If you only provide the table name parameter instead of a complete ARN, the API operation will be performed on the table in the account to which the requestor belongs. For an example of a policy that uses cross-account access, see Resource-based policy for cross-account access.

The resource owner's account will be charged even when a principal from another account is reading from or writing to the DynamoDB table in the owner's account. If the table has provisioned throughput, the sum of all the requests from the owner accounts and the requestors in other accounts will determine if the request will be throttled (if autoscaling is disabled) or scaled up/down if autoscaling is enabled.

The requests will be logged in the CloudTrail logs of both the owner and the requestor accounts so that each of the two accounts can track which account accessed what data.

Share access with cross-account AWS Lambda functions

Lambda functions in account A

1. Go to the IAM console to create an IAM role that will be used as the Lambda execution role for your AWS Lambda function in account A. Add the managed IAM policy `AWSLambdaDynamoDBExecutionRole` which has the required DynamoDB Streams and Lambda invocation permissions. This policy also grants access to all potential DynamoDB Streams resources you may have access to in account A.
2. In the Lambda console, create an AWS Lambda function to process records in a DynamoDB stream and during the setup for the execution role, choose the role you created in the previous step.
3. Provide the Lambda function execution role to the DynamoDB Streams' owner of account B to configure the resource-based policy for cross-account read access.
4. Finish setting up the Lambda function.

DynamoDB Stream in Account B

1. Get the cross-account Lambda execution role from account A that will invoke the Lambda function.
2. On the Amazon DynamoDB console in account B, choose the table for Lambda cross-account trigger. Under the **Exports and streams** tab, locate your DynamoDB stream ARN. Ensure that DynamoDB Stream status is On and note the full stream ARN as you will need it for the resource policy.
3. Under the **Permissions** tab, click the **create stream policy** button to start the visual policy editor. Click the **Add new statement** button or edit the policy if one already exists.
4. Create a policy that specifies the Lambda execution role in account A as the principal and grant the required DynamoDB Stream actions. Make sure to include the actions `dynamodb:DescribeStream`, `dynamodb:GetRecords`, `dynamodb:GetShardIterator`, and `dynamodb:ListShards`. For more information on example resource policies for DynamoDB Streams, see DynamoDB resource-based policy examples.

> **Note**
>
> The cross-account access of control plane APIs has a lower transactions per second (TPS) limit of 500 requests.

Blocking public access with resource-based policies in DynamoDB

Block Public Access (BPA) is a feature that identifies and prevents the attaching of resource-based policies that grant public access to your DynamoDB tables, indexes, or streams across your Amazon Web Services (AWS) accounts. With BPA, you can prevent public access to your DynamoDB resources. BPA performs checks during the creation or modification of a resource-based policy and helps improve your security posture with DynamoDB.

BPA uses automated reasoning to analyze the access granted by your resource-based policy and alerts you if such permissions are found at the time of administering a resource-based policy. The analysis verifies access across all resource-based policy statements, actions, and the set of condition keys used in your policies.

> **Important**
>
> BPA helps protect your resources by preventing public access from being granted through the resource-based policies that are directly attached to your DynamoDB resources, such as tables, indexes, and streams. In addition to using BPA, carefully inspect the following policies to confirm that they do not grant public access:
>
> - Identity-based policies attached to associated AWS principals (for example, IAM roles)
> - Resource-based policies attached to associated AWS resources (for example, AWS Key Management Service (KMS) keys)

You must ensure that the principal doesn't include a * entry or that one of the specified condition keys restrict access from principals to the resource. If the resource-based policy grants public access to your table, indexes, or stream across AWS accounts, DynamoDB will block you from creating or modifying the policy until the specification within the policy is corrected and deemed non-public.

You can make a policy non-public by specifying one or more principals inside the `Principal` block. The following resource-based policy example blocks public access by specifying two principals.

```
{
  "Effect": "Allow",
  "Principal": {
    "AWS": [
      "123456789012",
```

```
      "111122223333"
    ]
  },
  "Action": "dynamodb:*",
  "Resource": "*"
}
```

Policies that restrict access by specifying certain condition keys are also not considered public. Along with evaluation of the principal specified in the resource-based policy, the following trusted condition keys are used to complete the evaluation of a resource-based policy for non-public access:

- `aws:PrincipalAccount`
- `aws:PrincipalArn`
- `aws:PrincipalOrgID`
- `aws:PrincipalOrgPaths`
- `aws:SourceAccount`
- `aws:SourceArn`
- `aws:SourceVpc`
- `aws:SourceVpce`
- `aws:UserId`
- `aws:PrincipalServiceName`
- `aws:PrincipalServiceNamesList`
- `aws:PrincipalIsAWSService`
- `aws:Ec2InstanceSourceVpc`
- `aws:SourceOrgID`
- `aws:SourceOrgPaths`

Additionally, for a resource-based policy to be non-public, the values for Amazon Resource Name (ARN) and string keys must not contain wildcards or variables. If your resource-based policy uses the `aws:PrincipalIsAWSService` key, you must make sure that you've set the key value to true.

The following policy limits access to the user `John` in the specified account. The condition makes the `Principal` constrained and not be considered as public.

```
{
  "Effect": "Allow",
  "Principal": {
    "AWS": "*"
  },
  "Action": "dynamodb:*",
  "Resource": "*",
  "Condition": {
    "StringEquals": {
      "aws:PrincipalArn": "arn:aws:iam::123456789012:user/John"
    }
  }
}
```

The following example of a non-public resource-based policy constrains `sourceVPC` using the `StringEquals` operator.

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "AWS": "*"
      },
      "Action": "dynamodb:*",
      "Resource": "arn:aws:dynamodb:us-east-1:123456789012:table/
MusicCollection",
      "Condition": {
        "StringEquals": {
          "aws:SourceVpc": [
            "vpc-91237329"
          ]
        }
      }
    }
  ]
}
```

DynamoDB API operations supported by resource-based policies

This topic lists the API operations that are supported by resource-based policies. However, for cross-account access, you can only use a certain set of DynamoDB APIs through resource-based policies. You can't attach resource-based policies to resource types, such as backups and imports. The IAM actions, which correspond with the APIs operating on these resource types, are excluded from the supported IAM actions in resource-based policies. Because table administrators configure internal table settings within the same account, APIs, such as UpdateTimeToLive and DisableKinesisStreamingDestination, don't support cross-account access through resource-based policies.

The DynamoDB data plane and control plane APIs that support cross-account access also support table name overloading, which lets you specify the table ARN instead of the table name. You can specify table ARN in the `TableName` parameter of these APIs. However, not all of these APIs support cross-account access.

Topics

- Data plane API operations
- PartiQL API operations
- Control plane API operations
- Version 2019.11.21 (Current) global tables API operations
- Version 2017.11.29 (Legacy) global tables API operations
- Tags API operations
- Backup and Restore API operations
- Continuous Backup/Restore (PITR) API operations
- Contributor Insights API operations
- Export API operations
- Import API operations
- Amazon Kinesis Data Streams API operations
- Resource-based policy API operations
- Time-to-Live API operations
- Other API operations
- DynamoDB Streams API operations

Data plane API operations

The following table lists the API-level support provided by data plane API operations for resource-based policies and cross-account access.

Data Plane - Tables/indexes APIs	Resource-based policy support	Cross-account support
DeleteItem	Yes	Yes
GetItem	Yes	Yes
PutItem	Yes	Yes
Query	Yes	Yes
Scan	Yes	Yes
UpdateItem	Yes	Yes
TransactGetItems	Yes	Yes
TransactWriteItems	Yes	Yes
BatchGetItem	Yes	Yes
BatchWriteItem	Yes	Yes

PartiQL API operations

The following table lists the API-level support provided by PartiQL API operations for resource-based policies and cross-account access.

PartiQL APIs	Resource-based policy support	Cross-account support
BatchExecuteStatement	Yes	No
ExecuteStatement	Yes	No

PartiQL APIs	Resource-based policy support	Cross-account support
ExecuteTransaction	Yes	No

Control plane API operations

The following table lists the API-level support provided by control plane API operations for resource-based policies and cross-account access.

Control Plane - Tables APIs	Resource-based policy support	Cross-account support
CreateTable	No	No
DeleteTable	Yes	Yes
DescribeTable	Yes	Yes
UpdateTable	Yes	Yes

Version 2019.11.21 (Current) global tables API operations

The following table lists the API-level support provided by Version 2019.11.21 (Current) global tables API operations for resource-based policies and cross-account access.

Version 2019.11.21 (Current) global tables APIs	Resource-based policy support	Cross-account support
DescribeTableReplicaAutoScaling	Yes	No
UpdateTableReplicaAutoScaling	Yes	No

Version 2017.11.29 (Legacy) global tables API operations

The following table lists the API-level support provided by Version 2017.11.29 (Legacy) global tables API operations for resource-based policies and cross-account access.

Version 2017.11.29 (Legacy) global tables APIs	Resource-based policy support	Cross-account support
CreateGlobalTable	No	No
DescribeGlobalTable	No	No
DescribeGlobalTableSettings	No	No
ListGlobalTables	No	No
UpdateGlobalTable	No	No
UpdateGlobalTableSettings	No	No

Tags API operations

The following table lists the API-level support provided by API operations related to tags for resource-based policies and cross-account access.

Tags APIs	Resource-based policy support	Cross-account support
ListTagsOfResource	Yes	Yes
TagResource	Yes	Yes
UntagResource	Yes	Yes

Backup and Restore API operations

The following table lists the API-level support provided by API operations related to backup and restore for resource-based policies and cross-account access.

Backup and Restore APIs	Resource-based policy support	Cross-account support
CreateBackup	Yes	No
DescribeBackup	No	No
DeleteBackup	No	No
RestoreTableFromBackup	No	No

Continuous Backup/Restore (PITR) API operations

The following table lists the API-level support provided by API operations related to Continuous Backup/Restore (PITR) for resource-based policies and cross-account access.

Continuous Backup/Restore (PITR) APIs	Resource-based policy support	Cross-account support
DescribeContinuousBackups	Yes	No
RestoreTableToPointInTime	Yes	No
UpdateContinuousBackups	Yes	No

Contributor Insights API operations

The following table lists the API-level support provided by API operations related to Continuous Backup/Restore (PITR) for resource-based policies and cross-account access.

Contributor Insights APIs	Resource-based policy support	Cross-account support
DescribeContributorInsights	Yes	No
ListContributorInsights	No	No
UpdateContributorInsights	Yes	No

Export API operations

The following table lists the API-level support provided by Export API operations for resource-based policies and cross-account access.

Export APIs	Resource-based policy support	Cross-account support
DescribeExport	No	No
ExportTableToPointInTime	Yes	No
ListExports	No	No

Import API operations

The following table lists the API-level support provided by Import API operations for resource-based policies and cross-account access.

Import APIs	Resource-based policy support	Cross-account support
DescribeImport	No	No
ImportTable	No	No
ListImports	No	No

Amazon Kinesis Data Streams API operations

The following table lists the API-level support provided by Kinesis Data Streams API operations for resource-based policies and cross-account access.

Kinesis APIs	Resource-based policy support	Cross-account support
DescribeKinesisStreamingDestination	Yes	No
DisableKinesisStreamingDestination	Yes	No
EnableKinesisStreamingDestination	Yes	No
UpdateKinesisStreamingDestination	Yes	No

Resource-based policy API operations

The following table lists the API-level support provided by resource-based policy API operations for resource-based policies and cross-account access.

Resource-based policy APIs	Resource-based policy support	Cross-account support
GetResourcePolicy	Yes	No
PutResourcePolicy	Yes	No
DeleteResourcePolicy	Yes	No

Time-to-Live API operations

The following table lists the API-level support provided by time to live (TTL) API operations for resource-based policies and cross-account access.

TTL APIs	Resource-based policy support	Cross-account support
DescribeTimeToLive	Yes	No
UpdateTimeToLive	Yes	No

Other API operations

The following table lists the API-level support provided by other miscellaneous API operations for resource-based policies and cross-account access.

Other APIs	Resource-based policy support	Cross-account support
DescribeLimits	No	No
DescribeEndpoints	No	No
ListBackups	No	No
ListTables	No	No

DynamoDB Streams API operations

The following table lists the API-level support of DynamoDB Streams APIs for resource-based policies and cross-account access.

DynamoDB Streams APIs	Resource-based policy support	Cross-account support
DescribeStream	Yes	Yes
GetRecords	Yes	Yes
GetShardIterator	Yes	Yes
ListStreams	No	No

Authorization with IAM identity-based policies and DynamoDB resource-based policies

Identity-based policies are attached to an identity, such as IAM users, groups of users, and roles. These are IAM policy documents that control what actions an identity can perform, on which resources, and under what conditions. Identity-based policies can be managed or inline policies.

Resource-based policies are IAM policy documents that you attach to a resource, such as a DynamoDB table. These policies grant the specified principal permission to perform specific actions on that resource and defines under what conditions this applies. For example, the resource-based policy for a DynamoDB table also includes the index associated with the table. Resource-based policies are inline policies. There are no managed resource-based policies.

For more information about these policies, see Identity-based policies and resource-based policies in the *IAM User Guide*.

If the IAM principal is from the same account as the resource owner, a resource-based policy is sufficient to specify access permissions to the resource. You can still choose to have an IAM identity-based policy along with a resource-based policy. For cross-account access, you must explicitly allow access in both the identity and resource policies as specified in Cross-account access with resource-based policies in DynamoDB. When you use both types of policies, a policy is evaluated as described in Determining whether a request is allowed or denied within an account.

> **Important**
>
> If an identity-based policy grants unconditional access to a DynamoDB table (for example, `dynamodb:GetItem` with no conditions), a resource-based policy that allows access with conditions on `dynamodb:Attributes` won't restrict that access. The identity-based policy's unconditional allow takes precedence, and the resource-based policy's conditions are not applied as restrictions. To restrict access to specific attributes, use an explicit `Deny` statement instead of relying solely on conditional `Allow` statements in the resource-based policy.

DynamoDB resource-based policy examples

When you specify an ARN in the `Resource` field of a resource-based policy, the policy takes effect only if the specified ARN matches the ARN of the DynamoDB resource to which it is attached.

> **Note**
>
> Remember to replace the *italicized* text with your resource-specific information.

Resource-based policy for a table

The following resource-based policy attached to a DynamoDB table named *MusicCollection*, gives the IAM users *John* and *Jane* permission to perform GetItem and BatchGetItem actions on the *MusicCollection* resource.

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Sid": "1111",
        "Effect": "Allow",
        "Principal": {
          "AWS": [
            "arn:aws:iam::111122223333:user/username",
            "arn:aws:iam::111122223333:user/Jane"
          ]
        },
        "Action": [
          "dynamodb:GetItem",
          "dynamodb:BatchGetItem"
        ],
        "Resource": [
          "arn:aws:dynamodb:us-east-1:123456789012:table/MusicCollection"
        ]
    }
  ]
}
```

Resource-based policy for a stream

The following resource-based policy attached to a DynamoDB stream named `2024-02-12T18:57:26.492` gives the IAM users *John* and *Jane* permission to perform GetRecords, GetShardIterator, and DescribeStream API actions on the `2024-02-12T18:57:26.492` resource.

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Sid": "1111",
      "Effect": "Allow",
      "Principal": {
        "AWS": [
          "arn:aws:iam::111122223333:user/username",
          "arn:aws:iam::111122223333:user/Jane"
        ]
      },
      "Action": [
        "dynamodb:DescribeStream",
        "dynamodb:GetRecords",
        "dynamodb:GetShardIterator"
      ],
      "Resource": [
        "arn:aws:dynamodb:us-east-1:123456789012:table/MusicCollection/
stream/2024-02-12T18:57:26.492"
      ]
    }
  ]
}
```

Resource-based policy for access to perform all actions on specified resources

To allow a user to perform all actions on a table and all associated indexes with a table, you can use a wildcard (*) to represent the actions and the resources associated with the table. Using a

wild card character for the resources, will allow the user access to the DynamoDB table and all its associated indexes, including the ones that haven't yet been created. For example, the following policy will give the user *John* permission to perform any actions on the *MusicCollection* table and all of its indexes, including any indexes that will be created in the future.

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Sid": "1111",
      "Effect": "Allow",
      "Principal": {
        "AWS": "arn:aws:iam::111122223333:user/role-name"
      },
      "Action": "dynamodb:*",
      "Resource": [
        "arn:aws:dynamodb:us-east-1:123456789012:table/MusicCollection",
        "arn:aws:dynamodb:us-east-1:123456789012:table/MusicCollection/
index/index-name"
      ]
    }
  ]
}
```

Resource-based policy for cross-account access

You can specify permissions for a cross-account IAM identity to access DynamoDB resources. For example, you might need a user from a trusted account to get access to read the contents of your table, with the condition that they access only specific items and specific attributes in those items. The following policy allows access to user *John* from a trusted AWS account ID *111111111111* to access data from a table in account *123456789012* by using the GetItem API. The policy ensures that the user can access only items with a primary key *Jane* and that the user can only retrieve the attributes `Artist` and `SongTitle`, but no other attributes.

> **Important**
>
> If you do not specify the `SPECIFIC_ATTRIBUTES` condition, you'll see all attributes for the items returned.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "CrossAccountTablePolicy",
            "Effect": "Allow",
            "Principal": {
                "AWS": "arn:aws:iam::111111111111:user/John"
            },
            "Action": "dynamodb:GetItem",
            "Resource": [
                "arn:aws:dynamodb:us-west-2:123456789012:table/MusicCollection"
            ],
            "Condition": {
                "ForAllValues:StringEquals": {
                    "dynamodb:LeadingKeys": "Jane",
                    "dynamodb:Attributes": [
                        "Artist",
                        "SongTitle"
                    ]
                },
                "StringEquals": {
                    "dynamodb:Select": "SPECIFIC_ATTRIBUTES"
                }
            }
        }
    ]
}
```

In addition to the preceding resource-based policy, the identity-based policy attached to the user *John* also needs to allow the `GetItem` API action for the cross-account access to work. The following is an example of an identity-based policy that you must attach to the user *John*.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "CrossAccountIdentityBasedPolicy",
            "Effect": "Allow",
            "Action": [
                "dynamodb:GetItem"
            ],
            "Resource": [
                "arn:aws:dynamodb:us-east-1:123456789012:table/MusicCollection"
            ],
            "Condition": {
                "ForAllValues:StringEquals": {
                    "dynamodb:LeadingKeys": "Jane",
                    "dynamodb:Attributes": [
                        "Artist",
                        "SongTitle"
                    ]
                },
                "StringEquals": {
                    "dynamodb:Select": "SPECIFIC_ATTRIBUTES"
                }
            }
        }
    ]
}
```

The user John can make a `GetItem` request by specifying the table ARN in the `table-name` parameter for accessing the table *`MusicCollection`* in the account *`123456789012`*.

```
aws dynamodb get-item \
    --table-name arn:aws:dynamodb:us-west-2:123456789012:table/MusicCollection \
    --key '{"Artist": {"S": "Jane"}' \
    --projection-expression 'Artist, SongTitle' \
    --return-consumed-capacity TOTAL
```

Resource-based policy with IP address conditions

You can apply a condition to restrict source IP addresses, virtual private clouds (VPCs), and VPC endpoint (VPCE). You can specify permissions based on the source addresses of the originating request. For example, you might want to allow a user to access DynamoDB resources only if they are being accessed from a specific IP source, such as a corporate VPN endpoint. Specify these IP addresses in the `Condition` statement.

The following example allows the user *John* access to any DynamoDB resource when the source IPs are `54.240.143.0/24` and `2001:DB8:1234:5678::/64`.

JSON

```
{
  "Id":"PolicyId2",
  "Version":"2012-10-17",
  "Statement":[
    {
      "Sid":"AllowIPmix",
      "Effect":"Allow",
      "Principal": {
        "AWS": "arn:aws:iam::111111111111:user/username"
      },
      "Action":"dynamodb:*",
      "Resource":"*",
      "Condition": {
        "IpAddress": {
          "aws:SourceIp": [
            "54.240.143.0/24",
            "2001:DB8:1234:5678::/64"
          ]
        }
      }
    }
  ]
}
```

You can also deny all access to DynamoDB resources except when the source is a specific VPC endpoint, for example *vpce-1a2b3c4d*.

Important

When you use DAX with DynamoDB tables that have IP-based resource policies in IPv6-only environments, you must configure additional access rules. If your resource policy restricts access to the IPv4 address space `0.0.0.0/0` on tables, you must allow access for the IAM role associated with your DAX cluster. Add an `ArnNotEquals` condition to your policy to ensure DAX maintains access to your DynamoDB tables. For more information see, DAX and IPv6.

JSON

```
{
  "Id":"PolicyId",
  "Version":"2012-10-17",
  "Statement": [
    {
      "Sid": "AccessToSpecificVPCEOnly",
      "Principal": "*",
      "Action": "dynamodb:*",
      "Effect": "Deny",
      "Resource": "*",
      "Condition": {
        "StringNotEquals":{
          "aws:sourceVpce":"vpce-1a2b3c4d"
        }
      }
    }
  ]
}
```

Resource-based policy using an IAM role

You can also specify an IAM service role in the resource-based policy. IAM entities that assume this role are bounded by the permissible actions specified for the role and to the specific set of resources within the resource-based policy.

The following example allows an IAM entity to perform all DynamoDB actions on the *MusicCollection* and *MusicCollection* DynamoDB resources.

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Sid": "1111",
      "Effect": "Allow",
      "Principal": { "AWS": "arn:aws:iam::111122223333:role/role-name" },
      "Action": "dynamodb:*",
      "Resource": [
        "arn:aws:dynamodb:us-east-1:123456789012:table/MusicCollection",
        "arn:aws:dynamodb:us-east-1:123456789012:table/MusicCollection/*"
      ]
    }
  ]
}
```

DynamoDB resource-based policy considerations

When you define resource-based policies for your DynamoDB resources, the following considerations apply:

General considerations

- The maximum size supported for a resource-based policy document is 20 KB. DynamoDB counts whitespaces when calculating the size of a policy against this limit.
- Subsequent updates to a policy for a given resource are blocked for 15 seconds after a successful update of the policy for the same resource.
- Currently, you can only attach a resource-based policy to existing streams. You can't attach a policy to a stream while creating it.

Global table considerations

- Resource-based policies aren't supported for Global table version 2017.11.29 (Legacy) replicas.
- Within a resource-based policy, if the action for a DynamoDB service-linked role (SLR) to replicate data for a global table is denied, adding or deleting a replica will fail with an error.

- The AWS::DynamoDB::GlobalTable resource doesn't support creating a replica and adding a resource-based policy to that replica in the same stack update in Regions other than the Region where you deploy the stack update.

Cross-account considerations

- Cross-account access using resource-based policies doesn't support encrypted tables with AWS managed keys because you can't grant cross-account access to the AWS managed KMS policy.

CloudFormation considerations

- Resource-based policies don't support drift detection. If you update a resource-based policy outside of the AWS CloudFormation stack template, you'll need to update the CloudFormation stack with the changes.
- Resource-based policies don't support out of band changes. If you add, update, or delete a policy outside of the CloudFormation template, the change won't be overwritten if there are no changes to the policy within the template.

  For example, say that your template contains a resource-based policy which you later update outside of the template. If you don't make any changes to the policy in the template, the updated policy in DynamoDB won't be synced with the policy in the template.

  Conversely, say that your template doesn't contain a resource-based policy, but you add a policy outside of the template. This policy won't be removed from DynamoDB as long as you don't add it to the template. When you add a policy to the template and update the stack, the existing policy in DynamoDB will be updated to match the one defined in the template.

DynamoDB resource-based policy best practices

This topic describes the best practices for defining access permissions for your DynamoDB resources and the actions allowed on these resources.

Simplify access control to DynamoDB resources

If the AWS Identity and Access Management principals that need access to a DynamoDB resource are part of the same AWS account as the resource owner, an IAM identity-based policy is not required for each principal. A resource-based policy that is attached to the given resources will suffice. This type of configuration simplifies access control.

Protect your DynamoDB resources with resource-based policies

For all DynamoDB tables and streams, create resource-based policies to enforce access control for these resources. Resource-based policies enable you to centralize permissions at the resource level, simplify access control to DynamoDB tables, indexes, and streams, and reduce administration overhead. If no resource-based policy is specified for a table or a stream, access to the table or stream will be implicitly denied, unless identity-based policies associated with the IAM principals allow access.

Apply least-privilege permissions

When you set permissions with resource-based policies for DynamoDB resources, grant only the permissions required to perform an action. You do this by defining the actions that can be taken on specific resources under specific conditions, also known as least-privilege permissions. You might start with broad permissions while you explore the permissions that are required for your workload or use case. As your use case matures, you can work to reduce the permissions that you grant to work toward least privilege.

Analyze cross-account access activity for generating least-privilege policies

IAM Access Analyzer reports cross-account access to external entities specified in resource-based policies, and provides visibility to help you refine permissions and conform to least privilege. For more information about policy generation, see IAM Access Analyzer policy generation.

Use IAM Access Analyzer to generate least-privilege policies

To grant only the permissions required to perform a task, you can generate policies based on your access activity that is logged in AWS CloudTrail. IAM Access Analyzer analyzes the services and actions that your policies use.

Using attribute-based access control with DynamoDB

Attribute-based access control (ABAC) is an authorization strategy that defines access permissions based on tag conditions in your identity-based policies or other AWS policies, such as resource-based policies and organization IAM policies. You can attach tags to DynamoDB tables, which are then evaluated against the tag-based conditions. The indexes associated with a table inherit the tags you add to the table. You can add up to 50 tags for each DynamoDB table. The maximum size supported for all the tags in a table is 10 KB. For more information about tagging DynamoDB

resources and tagging restrictions, see Tagging resources in DynamoDB and the section called "Tagging restrictions in DynamoDB".

For more information about using tags to control access to AWS resources, see the following topics in the IAM User Guide:

- What is ABAC for AWS
- Controlling access to AWS resources using tags

Using ABAC, you can enforce different access levels for your teams and applications to perform actions on DynamoDB tables using fewer policies. You can specify a tag in the condition element of an IAM policy to control access to your DynamoDB tables or indexes. These conditions determine the level of access an IAM principal, a user, or role, has to DynamoDB tables and indexes. When an IAM principal makes an access request to DynamoDB, the resource and identity's tags are evaluated against the tag conditions in the IAM policy. Thereafter, the policy becomes effective only if tag conditions are met. This enables you to create an IAM policy that effectively says one of the following:

- *Allow the user to manage only those resources that have a tag with a key X and a value Y.*
- *Deny access to all users to resources tagged with a key X.*

For example, you can create a policy that allows users to update a table only if it has the tag key-value pair: `"environment": "staging"`. You can use the aws:ResourceTag condition key to allow or deny access to a table based on the tags that are attached to that table.

You can include attribute-based conditions while creating the policy or later using the AWS Management Console, AWS API, AWS Command Line Interface (AWS CLI), AWS SDK, or AWS CloudFormation.

The following example allows the UpdateItem action on a table named `MusicTable` if it includes a tag key with the name `environment` and value `production`.

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
```

```
    {
      "Effect": "Allow",
      "Action": [
        "dynamodb:UpdateItem"
      ],
      "Resource": "arn:aws:dynamodb:*:*:table/MusicTable",
      "Condition": {
        "StringEquals": {
          "aws:ResourceTag/environment": "production"
        }
      }
    }
  ]
}
```

Topics

- Why should I use ABAC?
- Condition keys to implement ABAC with DynamoDB
- Considerations for using ABAC with DynamoDB
- Enabling ABAC in DynamoDB
- Using ABAC with DynamoDB tables and indexes
- Examples for using ABAC with DynamoDB tables and indexes
- Troubleshooting common ABAC errors for DynamoDB tables and indexes

Why should I use ABAC?

- **Simpler policy management:** You use fewer policies because you don't have to create different policies to define the level of access for each IAM principal.
- **Scalable access control:** Scaling access control is easier with ABAC because you don't have to update your policies when you create new DynamoDB resources. You can use tags to authorize access to IAM principals that contain tags matching the resource's tags. You can onboard new IAM principals or DynamoDB resources and apply appropriate tags to automatically grant the necessary permissions without having to make any policy changes.
- **Fine-grained permission management:** It's a best practice to grant least privilege when you create policies. Using ABAC, you can create tags for the IAM principal, and use them to grant access to specific actions and resources that match the tags on the IAM principal.

- **Alignment with corporate directory:** You can map tags with existing employee attributes from your corporate directory to align your access control policies with your organizational structure.

Condition keys to implement ABAC with DynamoDB

You can use the following condition keys in your AWS policies to control the level of access to your DynamoDB tables and indexes:

- aws:ResourceTag/tag-key: Controls access based on whether or not the tag key-value pair on a DynamoDB table or index matches the tag key and value in a policy. This condition key is relevant to all APIs that operate on an existing table or index.

  The `dynamodb:ResourceTag` conditions are evaluated as if you didn't attach any tags to a resource.
- aws:RequestTag/tag-key: Allows comparing the tag key-value pair that was passed in the request with the tag pair that you specify in the policy. This condition key is relevant to APIs that contain tags as part of the request payload. These APIs include CreateTable and TagResource.
- aws:TagKeys: Compares the tag keys in a request with the keys that you specify in the policy. This condition key is relevant to APIs that contain tags as part of the request payload. These APIs include `CreateTable`, `TagResource`, and `UntagResource`.

Considerations for using ABAC with DynamoDB

When you use ABAC with DynamoDB tables or indexes, the following considerations apply:

- Tagging and ABAC aren't supported for DynamoDB Streams.
- Tagging and ABAC aren't supported for DynamoDB backups. To use ABAC with backups, we recommend that you use AWS Backup.
- Tags aren't preserved in restored tables. You need to add tags to restored tables before you can use tag-based conditions in your policies.

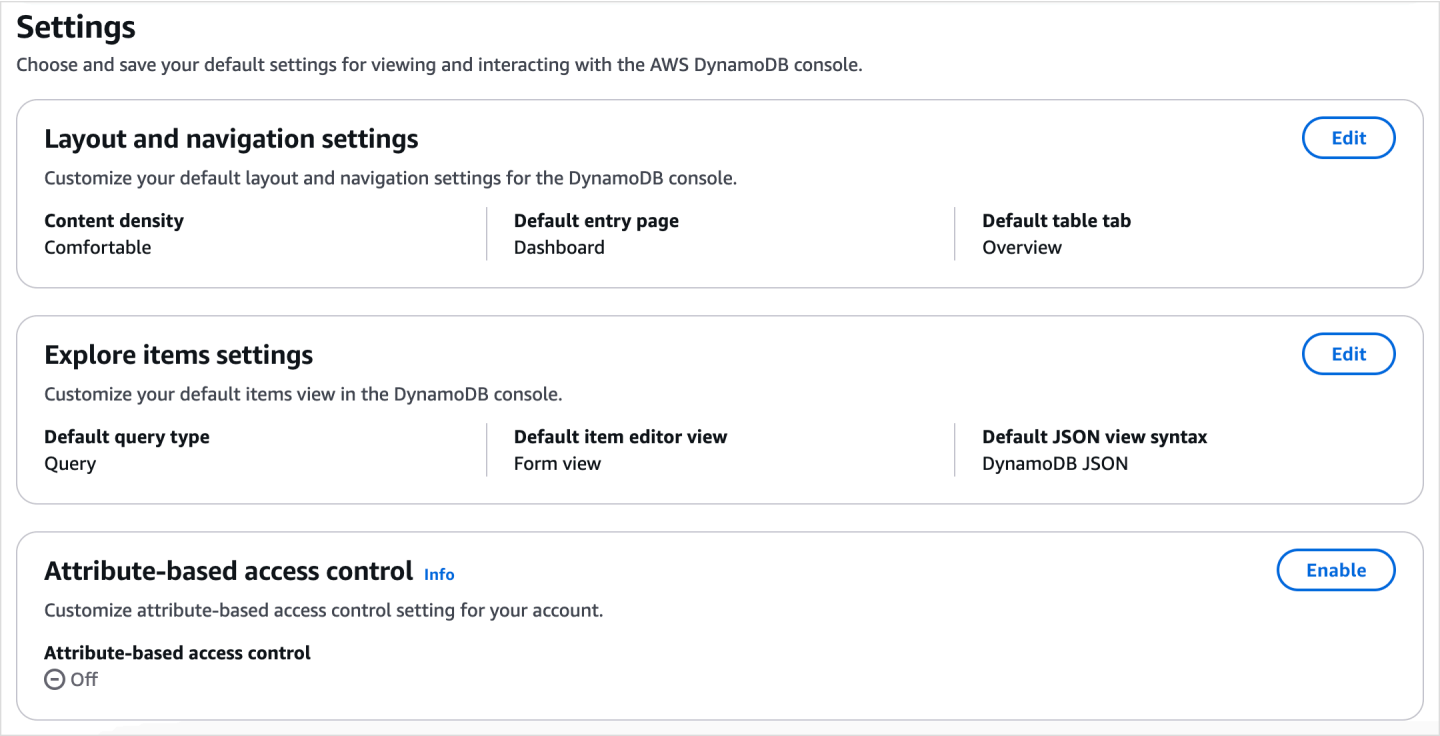
Enabling ABAC in DynamoDB

For most of the AWS accounts, ABAC is enabled by default. Using the DynamoDB console, you can confirm if ABAC is enabled for your account. To do this, make sure that you open the DynamoDB

console with a role that has the dynamodb:GetAbacStatus permission. Then, open the **Settings** page of the DynamoDB console.

If you don't see the **Attribute-based access control** card or if the card displays a status of **On**, it means ABAC is enabled for your account. However, if you see the **Attribute-based access control** card with a status of **Off**, as shown in the following image, ABAC isn't enabled for your account.

Attribute-based access control – not enabled



ABAC isn't enabled for AWS accounts for which tag-based conditions specified in their identity-based policies or other policies still need to be audited. If ABAC isn't enabled for your account, the tag-based conditions in your policies that are intended to act on DynamoDB tables or indexes are evaluated as if no tags are present for your resources or API requests. When ABAC is enabled for your account, the tag-based conditions in the policies of your account are evaluated considering the tags attached to your tables or API requests.

To enable ABAC for your account, we recommend that you first audit your policies as described in the Policy audit section. Then, include the required permissions for ABAC in your IAM policy. Finally, perform the steps described in Enabling ABAC in console to enable ABAC for your account in the current Region. After you enable ABAC, you can opt out within the next seven calendar days of opting in.

Topics

- Auditing your policies before enabling ABAC
- IAM permissions required to enable ABAC
- Enabling ABAC in console

Auditing your policies before enabling ABAC

Before you enable ABAC for your account, audit your policies to confirm that the tag-based conditions which might exist in the policies within your account are set up as intended. Auditing your policies will help avoid surprises from authorization changes with your DynamoDB workflows after ABAC is enabled. To view examples of using attribute-based conditions with tags, and the before and after behavior of ABAC implementation, see Examples for using ABAC with DynamoDB tables and indexes.

IAM permissions required to enable ABAC

You need the `dynamodb:UpdateAbacStatus` permission to enable ABAC for your account in the current Region. To confirm if ABAC is enabled for your account, you must also have the `dynamodb:GetAbacStatus` permission. With this permission, you can view the ABAC status for an account in any Region. You need these permissions in addition to the permission needed for accessing the DynamoDB console.

The following IAM policy grants the permission to enable ABAC and view its status for an account in the current Region.

```
{
"version": "2012-10-17",        &TCX5-2025-waiver;
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "dynamodb:UpdateAbacStatus",
                "dynamodb:GetAbacStatus"
             ],
            "Resource": "*"
        }
    ]
}
```

Enabling ABAC in console

1. Sign in to the AWS Management Console and open the DynamoDB console at https://console.aws.amazon.com/dynamodb/.
2. From the top navigation pane, choose the Region for which you want to enable ABAC.
3. On the left navigation pane, choose **Settings**.
4. On the **Settings** page, do the following:

   a. In the **Attribute-based access control** card, choose **Enable**.

   b. In the **Confirm attribute-based access control setting** box, choose **Enable** to confirm your choice.

   This enables ABAC for the current Region and the **Attribute-based access control** card shows the status of **On**.

   If you want to opt out after enabling ABAC on the console, you can do so within the next seven calendar days of opting in. To opt out, choose **Disable** in the **Attribute-based access control** card on the **Settings** page.

   > **Note**
   >
   > Updating the status of ABAC is an asynchronous operation. If the tags in your policies aren't evaluated right away, you might need to wait for some time because the application of the changes is eventually consistent.

Using ABAC with DynamoDB tables and indexes

The following steps show how to set up permissions using ABAC. In this example scenario, you'll add tags to a DynamoDB table and create an IAM role with a policy that includes tag-based conditions. Then, you'll test the allowed permissions on the DynamoDB table by matching the tag conditions.

Topics

- Step 1: Add tags to a DynamoDB table
- Step 2: Create an IAM role with a policy including tag-based conditions
- Step 3: Test allowed permissions

Step 1: Add tags to a DynamoDB table

You can add tags to new or existing DynamoDB tables using the AWS Management Console, AWS API, AWS Command Line Interface (AWS CLI), AWS SDK, or AWS CloudFormation. For example, the following tag-resource CLI command adds a tag to a table named `MusicTable`.

```
aws dynamodb tag-resource –resource-arn arn:aws:dynamodb:us-east-1:123456789012:table/
MusicTable –tags Key=environment,Value=staging
```

Step 2: Create an IAM role with a policy including tag-based conditions

Create an IAM policy using the aws:ResourceTag/tag-key condition key to compare the tag key-value pair that's specified in the IAM policy with the key-value pair that's attached to the table. The following example policy allows users to put or update items in tables if these tables contain the tag key-value pair: `"environment": "staging"`. If a table doesn't have the specified tag key-value pair, these actions are denied.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "dynamodb:PutItem",
                "dynamodb:UpdateItem"
            ],
            "Resource": "arn:aws:dynamodb:*:*:table/*",
            "Condition": {
                "StringEquals": {
                    "aws:ResourceTag/environment": "staging"
                }
            }
        }
    ]
}
```

Step 3: Test allowed permissions

1. Attach the IAM policy to a test user or role in your AWS account. Make sure that the IAM principal you use doesn't already have access to the DynamoDB table through a different policy.
2. Make sure that your DynamoDB table contains the `"environment"` tag key with a value of `"staging"`.
3. Perform the `dynamodb:PutItem` and `dynamodb:UpdateItem` actions on the tagged table. These actions should succeed if the `"environment": "staging"` tag key-value pair is present.

   If you perform these actions on a table that doesn't have the `"environment": "staging"` tag key-value pair, your request will fail with an `AccessDeniedException`.

You can also review the other sample use cases described in the following section to implement ABAC and perform more tests.

Examples for using ABAC with DynamoDB tables and indexes

The following examples depict some use cases to implement attribute-based conditions using tags.

Topics

- Example 1: Allow an action using aws:ResourceTag
- Example 2: Allow an action using aws:RequestTag
- Example 3: Deny an action using aws:TagKeys

Example 1: Allow an action using aws:ResourceTag

Using the `aws:ResourceTag/tag-key` condition key, you can compare the tag key-value pair that's specified in an IAM policy with the key-value pair that's attached in a DynamoDB table. For example, you can allow a specific action, such as PutItem, if the tag conditions match in an IAM policy and a table. To do this, perform the following steps:

Using the AWS CLI

1. Create a table. The following example uses the create-table AWS CLI command to create a table named `myMusicTable`.

```
aws dynamodb create-table \
  --table-name myMusicTable \
  --attribute-definitions AttributeName=id,AttributeType=S \
  --key-schema AttributeName=id,KeyType=HASH \
  --provisioned-throughput ReadCapacityUnits=5,WriteCapacityUnits=5 \
  --region us-east-1
```

2. Add a tag to this table. The following tag-resource AWS CLI command example adds the tag key-value pair `Title: ProductManager` to the `myMusicTable`.

```
aws dynamodb tag-resource --region us-east-1 --resource-arn arn:aws:dynamodb:us-east-1:123456789012:table/myMusicTable --tags Key=Title,Value=ProductManager
```

3. Create an inline policy and add it to a role which has the AmazonDynamoDBReadOnlyAccess AWS managed policy attached to it, as shown in the following example.

   JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": "dynamodb:PutItem",
      "Resource": "arn:aws:dynamodb:*:*:table/*",
      "Condition": {
        "StringEquals": {
          "aws:ResourceTag/Title": "ProductManager"
        }
      }
    }
  ]
}
```

   This policy allows the `PutItem` action on the table when the tag key and value attached to the table matches with the tags specified in the policy.

4. Assume the role with the policies described in Step 3.

5. Use the put-item AWS CLI command to put an item to the `myMusicTable`.

```
aws dynamodb put-item \
    --table-name myMusicTable --region us-east-1 \
    --item '{
        "id": {"S": "2023"},
        "title": {"S": "Happy Day"},
        "info": {"M": {
            "rating": {"N": "9"},
            "Artists": {"L": [{"S": "Acme Band"}, {"S": "No One You Know"}]},
            "release_date": {"S": "2023-07-21"}
        }}
    }'
```

6. Scan the table to verify if the item was added to the table.

```
aws dynamodb scan --table-name myMusicTable  --region us-east-1
```

Using the AWS SDK for Java 2.x

1. Create a table. The following example uses the CreateTable API to create a table named `myMusicTable`.

```
DynamoDbClient dynamoDB = DynamoDbClient.builder().region(region).build();
CreateTableRequest createTableRequest = CreateTableRequest.builder()
    .attributeDefinitions(
        Arrays.asList(
            AttributeDefinition.builder()
            .attributeName("id")
            .attributeType(ScalarAttributeType.S)
            .build()
        )
    )
    .keySchema(
        Arrays.asList(
            KeySchemaElement.builder()
            .attributeName("id")
            .keyType(KeyType.HASH)
            .build()
        )
    )
```

```
    .provisionedThroughput(ProvisionedThroughput.builder()
        .readCapacityUnits(5L)
        .writeCapacityUnits(5L)
        .build()
    )
    .tableName("myMusicTable")
    .build();

CreateTableResponse createTableResponse =
 dynamoDB.createTable(createTableRequest);
String tableArn = createTableResponse.tableDescription().tableArn();
String tableName = createTableResponse.tableDescription().tableName();
```

2. Add a tag to this table. The TagResource API in the following example adds the tag key-value pair `Title: ProductManager` to the `myMusicTable`.

```
TagResourceRequest tagResourceRequest = TagResourceRequest.builder()
    .resourceArn(tableArn)
    .tags(
        Arrays.asList(
            Tag.builder()
            .key("Title")
            .value("ProductManager")
            .build()
        )
    )
    .build();
dynamoDB.tagResource(tagResourceRequest);
```

3. Create an inline policy and add it to a role which has the AmazonDynamoDBReadOnlyAccess AWS managed policy attached to it, as shown in the following example.

   JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": "dynamodb:PutItem",
      "Resource": "arn:aws:dynamodb:*:*:table/*",
```

```
      "Condition": {
        "StringEquals": {
          "aws:ResourceTag/Title": "ProductManager"
        }
      }
    }
  ]
}
```

This policy allows the `PutItem` action on the table when the tag key and value attached to the table matches with the tags specified in the policy.

4. Assume the role with the policies described in Step 3.
5. Use the PutItem API to put an item to the `myMusicTable`.

```
HashMap<String, AttributeValue> info = new HashMap<>();
info.put("rating", AttributeValue.builder().s("9").build());
info.put("artists", AttributeValue.builder().ss(List.of("Acme Band","No One You
 Know").build());
info.put("release_date", AttributeValue.builder().s("2023-07-21").build());

HashMap<String, AttributeValue> itemValues = new HashMap<>();
itemValues.put("id", AttributeValue.builder().s("2023").build());
itemValues.put("title", AttributeValue.builder().s("Happy Day").build());
itemValues.put("info", AttributeValue.builder().m(info).build());


PutItemRequest putItemRequest = PutItemRequest.builder()
                .tableName(tableName)
                .item(itemValues)
                .build();
dynamoDB.putItem(putItemRequest);
```

6. Scan the table to verify if the item was added to the table.

```
ScanRequest scanRequest = ScanRequest.builder()
                .tableName(tableName)
                .build();

ScanResponse scanResponse = dynamoDB.scan(scanRequest);
```

Using the AWS SDK for Python (Boto3)

1. Create a table. The following example uses the CreateTable API to create a table named `myMusicTable`.

```
create_table_response = ddb_client.create_table(
    AttributeDefinitions=[
        {
            'AttributeName': 'id',
            'AttributeType': 'S'
        },
    ],
    TableName='myMusicTable',
    KeySchema=[
        {
            'AttributeName': 'id',
            'KeyType': 'HASH'
        },
    ],
        ProvisionedThroughput={
        'ReadCapacityUnits': 5,
        'WriteCapacityUnits': 5
    },
)

table_arn = create_table_response['TableDescription']['TableArn']
```

2. Add a tag to this table. The TagResource API in the following example adds the tag key-value pair `Title: ProductManager` to the `myMusicTable`.

```
tag_resouce_response = ddb_client.tag_resource(
    ResourceArn=table_arn,
    Tags=[
        {
            'Key': 'Title',
            'Value': 'ProductManager'
        },
    ]
)
```

3. Create an inline policy and add it to a role which has the AmazonDynamoDBReadOnlyAccess AWS managed policy attached to it, as shown in the following example.

   JSON

   ```
   {
       "Version":"2012-10-17",
       "Statement": [
           {
           "Effect": "Allow",
           "Action": "dynamodb:PutItem",
           "Resource": "arn:aws:dynamodb:*:*:table/*",
           "Condition": {
               "StringEquals": {
               "aws:ResourceTag/Title": "ProductManager"
               }
           }
           }
       ]
       }
   ```

   This policy allows the `PutItem` action on the table when the tag key and value attached to the table matches with the tags specified in the policy.

4. Assume the role with the policies described in Step 3.

5. Use the PutItem API to put an item to the `myMusicTable`.

   ```
   put_item_response = client.put_item(
       TableName = 'myMusicTable'
       Item = {
           'id': '2023',
           'title': 'Happy Day',
           'info': {
               'rating': '9',
               'artists': ['Acme Band','No One You Know'],
               'release_date': '2023-07-21'
           }
       }
   )
   ```

6. Scan the table to verify if the item was added to the table.

```
scan_response = client.scan(
    TableName='myMusicTable'
)
```

Without ABAC

If ABAC isn't enabled for your AWS account, the tag conditions in the IAM policy and the DynamoDB table aren't matched. Consequently, the `PutItem` action returns an `AccessDeniedException` because of the effect of the `AmazonDynamoDBReadOnlyAccess` policy.

```
An error occurred (AccessDeniedException) when calling the PutItem operation:
 User: arn:aws:sts::123456789012:assumed-role/DynamoDBReadOnlyAccess/Alice is
 not authorized to perform: dynamodb:PutItem on resource: arn:aws:dynamodb:us-
east-1:123456789012:table/myMusicTable because no identity-based policy allows the
 dynamodb:PutItem action.
```

With ABAC

If ABAC is enabled for your AWS account, the `put-item` action completes successfully and adds a new item to your table. This is because the inline policy on the table allows the `PutItem` action if the tag conditions in the IAM policy and the table match.

Example 2: Allow an action using aws:RequestTag

Using the aws:RequestTag/tag-key condition key, you can compare the tag key-value pair that's passed in your request with the tag pair that's specified in the IAM policy. For example, you can allow a specific action, such as `CreateTable`, using the `aws:RequestTag` if the tag conditions don't match. To do this, perform the following steps:

Using the AWS CLI

1. Create an inline policy and add it to a role which has the AmazonDynamoDBReadOnlyAccess AWS managed policy attached to it, as shown in the following example.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "dynamodb:CreateTable",
                "dynamodb:TagResource"
            ],
            "Resource": "arn:aws:dynamodb:*:*:table/*",
            "Condition": {
                "StringEquals": {
                    "aws:RequestTag/Owner": "John"
                }
            }
        }
    ]
}
```

2. Create a table that contains the tag key-value pair of `"Owner": "John"`.

```
aws dynamodb create-table \
--attribute-definitions AttributeName=ID,AttributeType=S \
--key-schema AttributeName=ID,KeyType=HASH  \
--provisioned-throughput ReadCapacityUnits=1000,WriteCapacityUnits=500 \
--region us-east-1 \
--tags Key=Owner,Value=John \
--table-name myMusicTable
```

Using the AWS SDK for Python (Boto3)

1. Create an inline policy and add it to a role which has the AmazonDynamoDBReadOnlyAccess AWS managed policy attached to it, as shown in the following example.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "dynamodb:CreateTable",
                "dynamodb:TagResource"
            ],
            "Resource": "arn:aws:dynamodb:*:*:table/*",
            "Condition": {
                "StringEquals": {
                    "aws:RequestTag/Owner": "John"
                }
            }
        }
    ]
}
```

2. Create a table that contains the tag key-value pair of `"Owner": "John"`.

```
ddb_client = boto3.client('dynamodb')

create_table_response = ddb_client.create_table(
    AttributeDefinitions=[
        {
            'AttributeName': 'id',
            'AttributeType': 'S'
        },
    ],
    TableName='myMusicTable',
    KeySchema=[
        {
            'AttributeName': 'id',
            'KeyType': 'HASH'
        },
    ],
        ProvisionedThroughput={
        'ReadCapacityUnits': 1000,
        'WriteCapacityUnits': 500
```

```
    },
    Tags=[
        {
            'Key': 'Owner',
            'Value': 'John'
        },
    ],
)
```

Without ABAC

If ABAC isn't enabled for your AWS account, the tag conditions in the inline policy and the DynamoDB table aren't matched. Consequently, the `CreateTable` request fails and your table isn't created.

```
An error occurred (AccessDeniedException) when calling the CreateTable operation:
 User: arn:aws:sts::123456789012:assumed-role/Admin/John is not authorized to perform:
 dynamodb:CreateTable on resource: arn:aws:dynamodb:us-east-1:123456789012:table/
myMusicTable because no identity-based policy allows the dynamodb:CreateTable action.
```

With ABAC

If ABAC is enabled for your AWS account, your table creation request completes successfully. Because the tag key-value pair of `"Owner": "John"` is present in the `CreateTable` request, the inline policy allows the user `John` to perform the `CreateTable` action.

Example 3: Deny an action using aws:TagKeys

Using the aws:TagKeys condition key, you can compare the tag keys in a request with the keys that are specified in the IAM policy. For example, you can deny a specific action, such as `CreateTable`, using `aws:TagKeys` if a specific tag key is *not* present in the request. To do this, perform the following steps:

Using the AWS CLI

1. Add a customer managed policy to a role which has the AmazonDynamoDBFullAccess AWS managed policy attached to it, as shown in the following example.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Deny",
            "Action": [
                "dynamodb:CreateTable",
                "dynamodb:TagResource"
            ],
            "Resource": "arn:aws:dynamodb:*:*:table/*",
            "Condition": {
                "Null": {
                    "aws:TagKeys": "false"
                },
                "ForAllValues:StringNotEquals": {
                    "aws:TagKeys": "CostCenter"
                }
            }
        }
    ]
}
```

2. Assume the role to which the policy was attached, and create a table with the tag key `Title`.

```
aws dynamodb create-table \
--attribute-definitions AttributeName=ID,AttributeType=S \
--key-schema AttributeName=ID,KeyType=HASH  \
--provisioned-throughput ReadCapacityUnits=1000,WriteCapacityUnits=500 \
--region us-east-1 \
--tags Key=Title,Value=ProductManager \
--table-name myMusicTable
```

Using the AWS SDK for Python (Boto3)

1. Add a customer managed policy to a role which has the AmazonDynamoDBFullAccess AWS managed policy attached to it, as shown in the following example.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Deny",
            "Action": [
                "dynamodb:CreateTable",
                "dynamodb:TagResource"
            ],
            "Resource": "arn:aws:dynamodb:*:*:table/*",
            "Condition": {
                "Null": {
                    "aws:TagKeys": "false"
                },
                "ForAllValues:StringNotEquals": {
                    "aws:TagKeys": "CostCenter"
                }
            }
        }
    ]
}
```

2. Assume the role to which the policy was attached, and create a table with the tag key `Title`.

```
ddb_client = boto3.client('dynamodb')

create_table_response = ddb_client.create_table(
    AttributeDefinitions=[
        {
            'AttributeName': 'id',
            'AttributeType': 'S'
        },
    ],
    TableName='myMusicTable',
    KeySchema=[
        {
            'AttributeName': 'id',
            'KeyType': 'HASH'
```

```
        },
    ],
        ProvisionedThroughput={
        'ReadCapacityUnits': 1000,
        'WriteCapacityUnits': 500
    },
    Tags=[
        {
            'Key': 'Title',
            'Value': 'ProductManager'
        },
    ],
)
```

Without ABAC

If ABAC isn't enabled for your AWS account, DynamoDB doesn't send the tag keys in the `create-table` command to IAM. The `Null` condition ensures that the condition evaluates to `false` if there are no tag keys in the request. Because the `Deny` policy doesn't match, the `create-table` command completes successfully.

With ABAC

If ABAC is enabled for your AWS account, the tag keys passed in the `create-table` command are passed to IAM. The tag key `Title` is evaluated against the condition-based tag key, `CostCenter`, present in the `Deny` policy. The tag key `Title` doesn't match the tag key present in the `Deny` policy because of the `StringNotEquals` operator. Therefore, the `CreateTable` action fails and your table isn't created. Running the `create-table` command returns an `AccessDeniedException`.

```
An error occurred (AccessDeniedException) when calling the CreateTable operation:
 User: arn:aws:sts::123456789012:assumed-role/DynamoFullAccessRole/ProductManager is
 not authorized to perform: dynamodb:CreateTable on resource: arn:aws:dynamodb:us-
east-1:123456789012:table/myMusicTable with an explicit deny in an identity-based
 policy.
```

Troubleshooting common ABAC errors for DynamoDB tables and indexes

This topic provides troubleshooting advice for common errors and issues that you might encounter while implementing ABAC in DynamoDB tables or indexes.

Service-specific condition keys in policies result in an error

Service-specific condition keys aren't considered as valid condition keys. If you've used such keys in your policies, these will result in an error. To fix this issue, you must replace the service-specific condition keys with an appropriate condition key to implement ABAC in DynamoDB.

For example, say that you've used the `dynamodb:ResourceTag` condition key in an inline policy that performs the PutItem request. Imagine that the request fails with an `AccessDeniedException`. The following example shows the erroneous inline policy with the `dynamodb:ResourceTag` condition key.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "dynamodb:PutItem"
            ],
            "Resource": "arn:aws:dynamodb:*:*:table/*",
            "Condition": {
                "StringEquals": {
                    "dynamodb:ResourceTag/Owner": "John"
                }
            }
        }
    ]
}
```

To fix this issue, replace the `dynamodb:ResourceTag` condition key with `aws:ResourceTag`, as shown in the following example.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "dynamodb:PutItem"
            ],
            "Resource": "arn:aws:dynamodb:*:*:table/*",
            "Condition": {
                "StringEquals": {
                    "aws:ResourceTag/Owner": "John"
                }
            }
        }
    ]
}
```

Unable to opt out of ABAC

If ABAC was enabled for your account through Support, you won't be able to opt out of ABAC through the DynamoDB console. To opt out, contact Support.

You can opt out of ABAC yourself *only if* the following are true:

- You used the self-service way of opting in through the DynamoDB console.
- You're opting out within seven calendar days of opting in.

Data protection in DynamoDB

Amazon DynamoDB provides a highly durable storage infrastructure designed for mission-critical and primary data storage. Data is redundantly stored on multiple devices across multiple facilities in an Amazon DynamoDB Region.

DynamoDB protects user data stored at rest and also data in transit between on-premises clients and DynamoDB, and between DynamoDB and other AWS resources within the same AWS Region.

Topics

- DynamoDB encryption at rest
- Securing DynamoDB connections using VPC endpoints and IAM policies"

DynamoDB encryption at rest

All user data stored in Amazon DynamoDB is fully encrypted at rest. DynamoDB encryption at rest provides enhanced security by encrypting all your data at rest using encryption keys stored in AWS Key Management Service (AWS KMS). This functionality helps reduce the operational burden and complexity involved in protecting sensitive data. With encryption at rest, you can build security-sensitive applications that meet strict encryption compliance and regulatory requirements.

DynamoDB encryption at rest provides an additional layer of data protection by always securing your data in an encrypted table—including its primary key, local and global secondary indexes, streams, global tables, backups, and DynamoDB Accelerator (DAX) clusters whenever the data is stored in durable media. Organizational policies, industry or government regulations, and compliance requirements often require the use of encryption at rest to increase the data security of your applications. For more information about encryption for database applications, see AWS Database Encryption SDK.

Encryption at rest integrates with AWS KMS for managing the encryption keys that are used to encrypt your tables. For more information about key types and states, see AWS Key Management Service concepts in the *AWS Key Management Service Developer Guide*.

When creating a new table, you can choose one of the following AWS KMS key types to encrypt your table. You can switch between these key types at any time.

- **AWS owned key –** Default encryption type. The key is owned by DynamoDB (no additional charge).
- **AWS managed key –** The key is stored in your account and is managed by AWS KMS (AWS KMS charges apply).
- **Customer managed key –** The key is stored in your account and is created, owned, and managed by you. You have full control over the KMS key (AWS KMS charges apply).

For more information about key types, see Customer keys and AWS keys.

Note

- When creating a new DAX cluster with encryption at rest enabled, an AWS managed key will be used to encrypt data at rest in the cluster.
- If your table has a sort key, some of the sort keys that mark range boundaries are stored in plaintext in the table metadata.

When you access an encrypted table, DynamoDB decrypts the table data transparently. You don't have to change any code or applications to use or manage encrypted tables. DynamoDB continues to deliver the same single-digit millisecond latency that you have come to expect, and all DynamoDB queries work seamlessly on your encrypted data.

You can specify an encryption key when you create a new table or switch the encryption keys on an existing table by using the AWS Management Console, AWS Command Line Interface (AWS CLI), or the Amazon DynamoDB API. To learn how, see Managing encrypted tables in DynamoDB.

Encryption at rest using the AWS owned key is offered at no additional charge. However, AWS KMS charges apply for an AWS managed key and for a customer managed key. For more information about pricing, see AWS KMS pricing.

DynamoDB encryption at rest is available in all AWS Regions, including the AWS China (Beijing) and AWS China (Ningxia) Regions and the AWS GovCloud (US) Regions. For more information, see DynamoDB encryption at rest: How it works and DynamoDB encryption at rest usage notes.

DynamoDB encryption at rest: How it works

Amazon DynamoDB encryption at rest encrypts your data using 256-bit Advanced Encryption Standard (AES-256), which helps secure your data from unauthorized access to the underlying storage.

Encryption at rest integrates with AWS Key Management Service (AWS KMS) for managing the encryption keys that are used to encrypt your tables.

Note

In May 2022, AWS KMS changed the rotation schedule for AWS managed keys from every three years (approximately 1,095 days) to every year (approximately 365 days).

New AWS managed keys are automatically rotated one year after they are created, and approximately every year thereafter.
Existing AWS managed keys are automatically rotated one year after their most recent rotation, and every year thereafter.

AWS owned keys

AWS owned keys are not stored in your AWS account. They are part of a collection of KMS keys that AWS owns and manages for use in multiple AWS accounts. AWS services can use AWS owned keys to protect your data. AWS owned keys used by DynamoDB are rotated every year (approximately 365 days).

You cannot view, manage, or use AWS owned keys, or audit their use. However, you do not need to do any work or change any programs to protect the keys that encrypt your data.

You are not charged a monthly fee or a usage fee for use of AWS owned keys, and they do not count against AWS KMS quotas for your account.

AWS managed keys

AWS managed keys are KMS keys in your account that are created, managed, and used on your behalf by an AWS service that is integrated with AWS KMS. You can view the AWS managed keys in your account, view their key policies, and audit their use in AWS CloudTrail logs. However, you cannot manage these KMS keys or change their permissions.

Encryption at rest automatically integrates with AWS KMS for managing the AWS managed keys for DynamoDB (`aws/dynamodb`) that are used to encrypt your tables. If an AWS managed key doesn't exist when you create your encrypted DynamoDB table, AWS KMS automatically creates a new key for you. This key is used with encrypted tables that are created in the future. AWS KMS combines secure, highly available hardware and software to provide a key management system scaled for the cloud.

For more information about managing permissions of the AWS managed key, see Authorizing use of the AWS managed key in the *AWS Key Management Service Developer Guide*.

Customer managed keys

Customer managed keys are KMS keys in your AWS account that you create, own, and manage. You have full control over these KMS keys, including establishing and maintaining their key policies,

IAM policies, and grants; enabling and disabling them; rotating their cryptographic material; adding tags; creating aliases that refer to them; and scheduling them for deletion. For more information about managing permissions of a customer managed key, see Customer managed keys.

When you specify a customer managed key as the table-level encryption key, the DynamoDB table, local and global secondary indexes, and streams are encrypted with the same customer managed key. On-demand backups are encrypted with the table-level encryption key that is specified at the time the backup is created. Updating the table-level encryption key does not change the encryption key that is associated with existing on-demand backups.

Setting the state of the customer managed key to disabled or scheduling it for deletion prevents all users and the DynamoDB service from being able to encrypt or decrypt data and to perform read and write operations on the table. DynamoDB must have access to your encryption key to ensure that you can continue to access your table and to prevent data loss.

If you disable your customer managed key or schedule it for deletion, your table status becomes **Inaccessible**. To ensure that you can continue working with the table, you must provide DynamoDB access to the specified encryption key within seven days. As soon as the service detects that your encryption key is inaccessible, DynamoDB sends you an email notification to alert you.

Note

- If your customer managed key remains inaccessible to the DynamoDB service for longer than seven days, the table is archived and can no longer be accessed. DynamoDB creates an on-demand backup of your table, and you are billed for it. You can use this on-demand backup to restore your data to a new table. To initiate the restore, the last customer managed key on the table must be enabled, and DynamoDB must have access to it.
- If your customer managed key that was used to encrypt a global table replica is inaccessible DynamoDB will remove this replica from the replication group. The replica will not be deleted and replication will stop from and to this region, 20 hours after detecting the customer managed key as inaccessible.

For more information, see enabling keys and deleting keys.

Notes on using AWS managed keys

Amazon DynamoDB can't read your table data unless it has access to the KMS key stored in your AWS KMS account. DynamoDB uses envelope encryption and key hierarchy to encrypt data. Your AWS KMS encryption key is used to encrypt the root key of this key hierarchy. For more information, see Envelope encryption in the *AWS Key Management Service Developer Guide*.

DynamoDB doesn't call AWS KMS for every DynamoDB operation. The key is refreshed once every 5 minutes per caller with active traffic.

Ensure that you have configured the SDK to reuse connections. Otherwise, you will experience latencies from DynamoDB having to reestablish new AWS KMS cache entries for each DynamoDB operation. In addition, you might potentially have to face higher AWS KMS and CloudTrail costs. For example, to do this using the Node.js SDK, you can create a new HTTPS agent with `keepAlive` turned on. For more information, see Configuring keepAlive in Node.js in the *AWS SDK for JavaScript Developer Guide*.

DynamoDB encryption at rest usage notes

Consider the following when you are using encryption at rest in Amazon DynamoDB.

All table data is encrypted

Server-side encryption at rest is enabled on all DynamoDB table data and cannot be disabled. You cannot encrypt only a subset of items in a table.

Encryption at rest only encrypts data while it is static (at rest) on a persistent storage media. If data security is a concern for data in transit or data in use, you might need to take additional measures:

- Data in transit: All your data in DynamoDB is encrypted in transit. By default, communications to and from DynamoDB use the HTTPS protocol, which protects network traffic by using Secure Sockets Layer (SSL)/Transport Layer Security (TLS) encryption.
- Data in use: Protect your data before sending it to DynamoDB using client-side encryption. For more information, see Client-side and server-side encryption in the *Amazon DynamoDB Encryption Client Developer Guide*.

You can use streams with encrypted tables. DynamoDB streams are always encrypted with a table-level encryption key. For more information, see Change data capture for DynamoDB Streams.

DynamoDB backups are encrypted, and the table that is restored from a backup also has encryption enabled. You can use the AWS owned key, AWS managed key, or customer managed key to encrypt your backup data. For more information, see Backup and restore for DynamoDB.

Local secondary indexes and global secondary indexes are encrypted using the same key as the base table.

Encryption types

> **Note**
>
> Customer managed keys are not supported in Global Table Version 2017. If you want to use a customer managed key in a DynamoDB Global Table, you need to upgrade the table to Global Table Version 2019 and then enable it.

On the AWS Management Console, the encryption type is `KMS` when you use the AWS managed key or customer managed key to encrypt your data. The encryption type is `DEFAULT` when you use the AWS owned key. In the Amazon DynamoDB API, the encryption type is `KMS` when you use the AWS managed key or customer managed key. In the absence of encryption type, your data is encrypted using the AWS owned key. You can switch between the AWS owned key, AWS managed key, and customer managed key at any given time. You can use the console, the AWS Command Line Interface (AWS CLI), or the Amazon DynamoDB API to switch the encryption keys.

Note the following limitations when using customer managed keys:

- You cannot use a customer managed key with DynamoDB Accelerator (DAX) clusters. For more information, see DAX encryption at rest.
- You can use a customer managed key to encrypt tables that use transactions. However, to ensure durability for propagation of transactions, a copy of the transaction request is temporarily stored by the service and encrypted using an AWS owned key. Committed data in your tables and secondary indexes is always encrypted at rest using your customer managed key.
- You can use a customer managed key to encrypt tables that use Contributor Insights. However, data that is transmitted to Amazon CloudWatch is encrypted with an AWS owned key.
- When you transition to a new customer managed key, be sure to keep the original key enabled until the process is complete. AWS will still need the original key to decrypt the data before encrypting it with the new key. The process will be complete when the table's SSEDescription

Status is ENABLED and the KMSMasterKeyArn of the new customer managed key is displayed. At this point the original key can be disabled or scheduled for deletion.

- Once the new customer managed key is displayed, the table and any new on-demand backups are encrypted with the new key.
- Any existing on-demand backups remain encrypted with the customer managed key that was used when those backups were created. You will need that same key to restore those backups. You can identify the key for the period when each backup was created by using the DescribeBackup API to view that backup's SSEDescription.
- If you disable your customer managed key or schedule it for deletion, any data in DynamoDB Streams is still subject to a 24-hour lifetime. Any unretrieved activity data is eligible for trimming when it is older than 24 hours.
- If you disable your customer managed key or schedule it for deletion, Time to Live (TTL) deletes continue for 30 minutes. These TTL deletes continue to be emitted to DynamoDB Streams and are subject to the standard trimming/retention interval.

For more information, see enabling keys and deleting keys.

Using KMS keys and data keys

The DynamoDB encryption at rest feature uses an AWS KMS key and a hierarchy of data keys to protect your table data. DynamoDB uses the same key hierarchy to protect DynamoDB streams, global tables, and backups when they are written to durable media.

We recommend that you plan your encryption strategy before implementing your table in DynamoDB. If you store sensitive or confidential data in DynamoDB, consider including client-side encryption in your plan. This way you can encrypt data as close as possible to its origin, and ensure its protection throughout its lifecycle. For more information see the DynamoDB encryption client documentation.

AWS KMS key

Encryption at rest protects your DynamoDB tables under an AWS KMS key. By default, DynamoDB uses an AWS owned key, a multi-tenant encryption key that is created and managed in a DynamoDB service account. But you can encrypt your DynamoDB tables under a customer managed key for DynamoDB (`aws/dynamodb`) in your AWS account. You can select a different KMS key for each table. The KMS key you select for a table is also used to encrypt its local and global secondary indexes, streams, and backups.

You select the KMS key for a table when you create or update the table. You can change the KMS key for a table at any time, either in the DynamoDB console or by using the UpdateTable operation. The process of switching keys is seamless and does not require downtime or degrade service.

> **Important**
>
> DynamoDB supports only symmetric KMS keys. You cannot use an asymmetric KMS key to encrypt your DynamoDB tables.

Use a customer managed key to get the following features:

- You create and manage the KMS key, including setting the key policies, IAM policies and grants to control access to the KMS key. You can enable and disable the KMS key, enable and disable automatic key rotation, and delete the KMS key when it is no longer in use.
- You can use a customer managed key with imported key material or a customer managed key in a custom key store that you own and manage.
- You can audit the encryption and decryption of your DynamoDB table by examining the DynamoDB API calls to AWS KMS in AWS CloudTrail logs.

Use the AWS managed key if you need any of the following features:

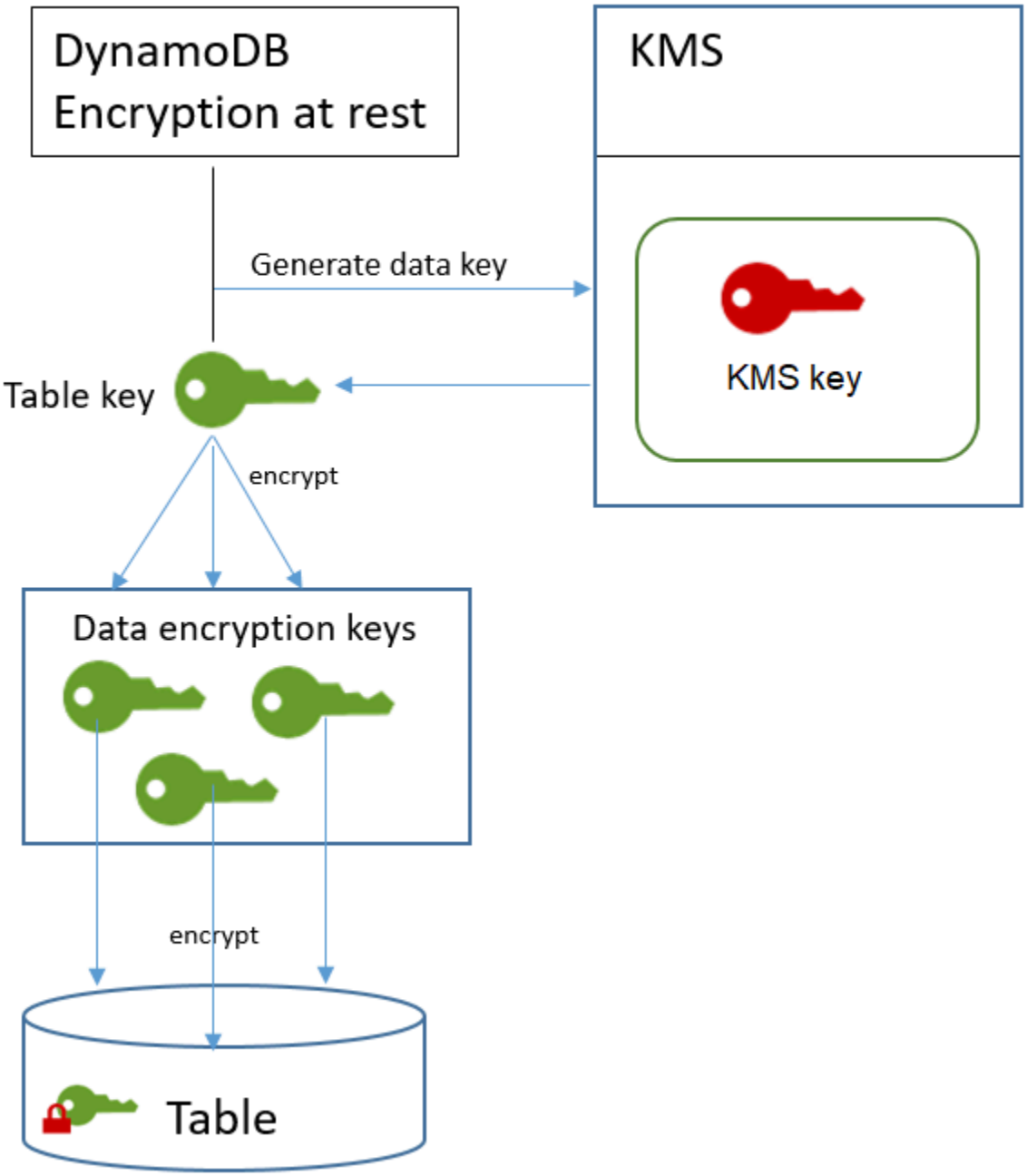
- You can view the KMS key and view its key policy. (You cannot change the key policy.)
- You can audit the encryption and decryption of your DynamoDB table by examining the DynamoDB API calls to AWS KMS in AWS CloudTrail logs.

However, the AWS owned key is free of charge and its use does not count against AWS KMS resource or request quotas. Customer managed keys and AWS managed keys incur a charge for each API call and AWS KMS quotas apply to these KMS keys.

Table keys

DynamoDB uses the KMS key for the table to generate and encrypt a unique data key for the table, known as the *table key*. The table key persists for the lifetime of the encrypted table.

The table key is used as a key encryption key. DynamoDB uses this table key to protect data encryption keys that are used to encrypt the table data. DynamoDB generates a unique data encryption key for each underlying structure in a table, but multiple table items might be protected by the same data encryption key.



When you first access an encrypted table, DynamoDB sends a request to AWS KMS to use the KMS key to decrypt the table key. Then, it uses the plaintext table key to decrypt the data encryption keys, and uses the plaintext data encryption keys to decrypt table data.

DynamoDB stores and uses the table key and data encryption keys outside of AWS KMS. It protects all keys with Advanced Encryption Standard (AES) encryption and 256-bit encryption

keys. Then, it stores the encrypted keys with the encrypted data so they are available to decrypt the table data on demand.

If you change the KMS key for your table, DynamoDB generates a new table key. Then, it uses the new table key to re-encrypt the data encryption keys.

Table key caching

To avoid calling AWS KMS for every DynamoDB operation, DynamoDB caches the plaintext table keys for each caller in memory. If DynamoDB gets a request for the cached table key after five minutes of inactivity, it sends a new request to AWS KMS to decrypt the table key. This call will capture any changes made to the access policies of the KMS key in AWS KMS or AWS Identity and Access Management (IAM) since the last request to decrypt the table key.

Authorizing use of your KMS key

If you use a customer managed key or the AWS managed key in your account to protect your DynamoDB table, the policies on that KMS key must give DynamoDB permission to use it on your behalf. The authorization context on the AWS managed key for DynamoDB includes its key policy and grants that delegate the permissions to use it.

You have full control over the policies and grants on a customer managed key Because the AWS managed key is in your account, you can view its policies and grants. But, because it is managed by AWS, you cannot change the policies.

DynamoDB does not need additional authorization to use the default AWS owned key to protect the DynamoDB tables in your AWS account.

Topics

- Key policy for an AWS managed key
- Key policy for a customer managed key
- Using grants to authorize DynamoDB

Key policy for an AWS managed key

When DynamoDB uses the AWS managed key for DynamoDB (`aws/dynamodb`) in cryptographic operations, it does so on behalf of the user who is accessing the DynamoDB resource. The key policy on the AWS managed key gives all users in the account permission to use the AWS managed key for specified operations. But permission is granted only when DynamoDB makes the request on

the user's behalf. The ViaService condition in the key policy does not allow any user to use the AWS managed key unless the request originates with the DynamoDB service.

This key policy, like the policies of all AWS managed keys, is established by AWS. You cannot change it, but you can view it at any time. For details, see Viewing a key policy.

The policy statements in the key policy have the following effect:

- Allow users in the account to use the AWS managed key for DynamoDB in cryptographic operations when the request comes from DynamoDB on their behalf. The policy also allows users to create grants for the KMS key.
- Allows authorized IAM identities in the account to view the properties of the AWS managed key for DynamoDB and to revoke the grant that allows DynamoDB to use the KMS key. DynamoDB uses grants for ongoing maintenance operations.
- Allows DynamoDB to perform read-only operations to find the AWS managed key for DynamoDB in your account.

JSON

```
{
  "Version":"2012-10-17",
  "Id" : "auto-dynamodb-1",
  "Statement" : [ {
    "Sid" : "Allow access through Amazon DynamoDB for all principals in the
 account that are authorized to use Amazon DynamoDB",
    "Effect" : "Allow",
    "Principal" : {
      "AWS" : "*"
    },
    "Action" : [ "kms:Encrypt", "kms:Decrypt", "kms:ReEncrypt*",
 "kms:GenerateDataKey*", "kms:CreateGrant", "kms:DescribeKey" ],
    "Resource" : "*",
    "Condition" : {
      "StringEquals" : {
        "kms:CallerAccount" : "111122223333",
        "kms:ViaService" : "dynamodb.us-west-2.amazonaws.com"
      }
    }
  }, {
    "Sid" : "Allow direct access to key metadata to the account",
```

```
    "Effect" : "Allow",
    "Principal" : {
      "AWS" : "arn:aws:iam::111122223333:root"
    },
    "Action" : [ "kms:Describe*", "kms:Get*", "kms:List*", "kms:RevokeGrant" ],
    "Resource" : "*"
  }, {
    "Sid" : "Allow DynamoDB Service with service principal name
 dynamodb.amazonaws.com to describe the key directly",
    "Effect" : "Allow",
    "Principal" : {
      "Service" : "dynamodb.amazonaws.com"
    },
    "Action" : [ "kms:Describe*", "kms:Get*", "kms:List*" ],
    "Resource" : "*"
  } ]
}
```

Key policy for a customer managed key

When you select a customer managed key to protect a DynamoDB table, DynamoDB gets permission to use the KMS key on behalf of the principal who makes the selection. That principal, a user or role, must have the permissions on the KMS key that DynamoDB requires. You can provide these permissions in a key policy, an IAM policy, or a grant.

At a minimum, DynamoDB requires the following permissions on a customer managed key:

- kms:Encrypt
- kms:Decrypt
- kms:ReEncrypt* (for kms:ReEncryptFrom and kms:ReEncryptTo)
- kms:GenerateDataKey* (for kms:GenerateDataKey and kms:GenerateDataKeyWithoutPlaintext)
- kms:DescribeKey
- kms:CreateGrant

For example, the following example key policy provides only the required permissions. The policy has the following effects:

- Allows DynamoDB to use the KMS key in cryptographic operations and create grants, but only when it is acting on behalf of principals in the account who have permission to use DynamoDB. If

the principals specified in the policy statement don't have permission to use DynamoDB, the call fails, even when it comes from the DynamoDB service.

- The kms:ViaService condition key allows the permissions only when the request comes from DynamoDB on behalf of the principals listed in the policy statement. These principals can't call these operations directly. Note that the `kms:ViaService` value, `dynamodb.*.amazonaws.com`, has an asterisk (*) in the Region position. DynamoDB requires the permission to be independent of any particular AWS Region so it can make cross-Region calls to support DynamoDB global tables.
- Gives the KMS key administrators (users who can assume the `db-team` role) read-only access to the KMS key and permission to revoke grants, including the grants that DynamoDB requires to protect the table.

Before using an example key policy, replace the example principals with actual principals from your AWS account.

JSON

```
{
  "Id": "key-policy-dynamodb",
  "Version":"2012-10-17",
  "Statement": [
    {
      "Sid" : "Allow access through Amazon DynamoDB for all principals in the
 account that are authorized to use Amazon DynamoDB",
      "Effect": "Allow",
      "Principal": {"AWS": "arn:aws:iam::111122223333:user/db-lead"},
      "Action": [
        "kms:Encrypt",
        "kms:Decrypt",
        "kms:ReEncrypt*",
        "kms:GenerateDataKey*",
        "kms:DescribeKey",
        "kms:CreateGrant"
      ],
      "Resource": "*",
      "Condition": {
         "StringLike": {
           "kms:ViaService" : "dynamodb.*.amazonaws.com"
         }
```

```
      }
    },
    {
      "Sid":  "Allow administrators to view the KMS key and revoke grants",
      "Effect": "Allow",
      "Principal": {
        "AWS": "arn:aws:iam::111122223333:role/db-team"
       },
      "Action": [
        "kms:Describe*",
        "kms:Get*",
        "kms:List*",
        "kms:RevokeGrant"
      ],
      "Resource": "*"
    }
  ]
}
```

Using grants to authorize DynamoDB

In addition to key policies, DynamoDB uses grants to set permissions on a customer managed key or the AWS managed key for DynamoDB (`aws/dynamodb`). To view the grants on a KMS key in your account, use the ListGrants operation. DynamoDB does not need grants, or any additional permissions, to use the AWS owned key to protect your table.

DynamoDB uses the grant permissions when it performs background system maintenance and continuous data protection tasks. It also uses grants to generate table keys.

Each grant is specific to a table. If the account includes multiple tables encrypted under the same KMS key, there is a grant of each type for each table. The grant is constrained by the DynamoDB encryption context, which includes the table name and the AWS account ID, and it includes permission to the retire the grant if it is no longer needed.

To create the grants, DynamoDB must have permission to call `CreateGrant` on behalf of the user who created the encrypted table. For AWS managed keys, DynamoDB gets `kms:CreateGrant` permission from the key policy, which allows account users to call CreateGrant on the KMS key only when DynamoDB makes the request on an authorized user's behalf.

The key policy can also allow the account to revoke the grant on the KMS key. However, if you revoke the grant on an active encrypted table, DynamoDB will not be able to protect and maintain the table.

DynamoDB encryption context

An encryption context is a set of key–value pairs that contain arbitrary nonsecret data. When you include an encryption context in a request to encrypt data, AWS KMS cryptographically binds the encryption context to the encrypted data. To decrypt the data, you must pass in the same encryption context.

DynamoDB uses the same encryption context in all AWS KMS cryptographic operations. If you use a customer managed key or an AWS managed key to protect your DynamoDB table, you can use the encryption context to identify use of the KMS key in audit records and logs. It also appears in plaintext in logs, such as AWS CloudTrail and Amazon CloudWatch Logs.

The encryption context can also be used as a condition for authorization in policies and grants. DynamoDB uses the encryption context to constrain the grants that allow access to the customer managed key or AWS managed key in your account and region.

In its requests to AWS KMS, DynamoDB uses an encryption context with two key–value pairs.

```
"encryptionContextSubset": {
    "aws:dynamodb:tableName": "Books"
    "aws:dynamodb:subscriberId": "111122223333"
}
```

- **Table** – The first key–value pair identifies the table that DynamoDB is encrypting. The key is `aws:dynamodb:tableName`. The value is the name of the table.

  ```
  "aws:dynamodb:tableName": "<table-name>"
  ```

  For example:

  ```
  "aws:dynamodb:tableName": "Books"
  ```

- **Account** – The second key–value pair identifies the AWS account. The key is `aws:dynamodb:subscriberId`. The value is the account ID.

  ```
  "aws:dynamodb:subscriberId": "<account-id>"
  ```

For example:

```
"aws:dynamodb:subscriberId": "111122223333"
```

Monitoring DynamoDB interaction with AWS KMS

If you use a customer managed key or an AWS managed key to protect your DynamoDB tables, you can use AWS CloudTrail logs to track the requests that DynamoDB sends to AWS KMS on your behalf.

The `GenerateDataKey`, `Decrypt`, and `CreateGrant` requests are discussed in this section. In addition, DynamoDB uses a DescribeKey operation to determine whether the KMS key you selected exists in the account and region. It also uses a RetireGrant operation to remove a grant when you delete a table.

GenerateDataKey

When you enable encryption at rest on a table, DynamoDB creates a unique table key. It sends a GenerateDataKey request to AWS KMS that specifies the KMS key for the table.

The event that records the `GenerateDataKey` operation is similar to the following example event. The user is the DynamoDB service account. The parameters include the Amazon Resource Name (ARN) of the KMS key, a key specifier that requires a 256-bit key, and the encryption context that identifies the table and the AWS account.

```
{
    "eventVersion": "1.05",
    "userIdentity": {
        "type": "AWSService",
        "invokedBy": "dynamodb.amazonaws.com"
    },
    "eventTime": "2018-02-14T00:15:17Z",
    "eventSource": "kms.amazonaws.com",
    "eventName": "GenerateDataKey",
    "awsRegion": "us-west-2",
    "sourceIPAddress": "dynamodb.amazonaws.com",
    "userAgent": "dynamodb.amazonaws.com",
    "requestParameters": {
        "encryptionContext": {
            "aws:dynamodb:tableName": "Services",
```

```
            "aws:dynamodb:subscriberId": "111122223333"
        },
        "keySpec": "AES_256",
        "keyId": "1234abcd-12ab-34cd-56ef-1234567890ab"
    },
    "responseElements": null,
    "requestID": "229386c1-111c-11e8-9e21-c11ed5a52190",
    "eventID": "e3c436e9-ebca-494e-9457-8123a1f5e979",
    "readOnly": true,
    "resources": [
        {
            "ARN": "arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab",
            "accountId": "111122223333",
            "type": "AWS::KMS::Key"
        }
    ],
    "eventType": "AwsApiCall",
    "recipientAccountId": "111122223333",
    "sharedEventID": "bf915fa6-6ceb-4659-8912-e36b69846aad"
}
```

Decrypt

When you access an encrypted DynamoDB table, DynamoDB needs to decrypt the table key so that it can decrypt the keys below it in the hierarchy. It then decrypts the data in the table. To decrypt the table key. DynamoDB sends a Decrypt request to AWS KMS that specifies the KMS key for the table.

The event that records the `Decrypt` operation is similar to the following example event. The user is the principal in your AWS account who is accessing the table. The parameters include the encrypted table key (as a ciphertext blob) and the encryption context that identifies the table and the AWS account. AWS KMS derives the ID of the KMS key from the ciphertext.

```
{
    "eventVersion": "1.05",
    "userIdentity": {
        "type": "AssumedRole",
        "principalId": "AROAIGDTESTANDEXAMPLE:user01",
        "arn": "arn:aws:sts::111122223333:assumed-role/Admin/user01",
        "accountId": "111122223333",
        "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
        "sessionContext": {
```

```
            "attributes": {
                "mfaAuthenticated": "false",
                "creationDate": "2018-02-14T16:42:15Z"
            },
            "sessionIssuer": {
                "type": "Role",
                "principalId": "AROAIGDT3HGFQZX4RY6RU",
                "arn": "arn:aws:iam::111122223333:role/Admin",
                "accountId": "111122223333",
                "userName": "Admin"
            }
        },
        "invokedBy": "dynamodb.amazonaws.com"
    },
    "eventTime": "2018-02-14T16:42:39Z",
    "eventSource": "kms.amazonaws.com",
    "eventName": "Decrypt",
    "awsRegion": "us-west-2",
    "sourceIPAddress": "dynamodb.amazonaws.com",
    "userAgent": "dynamodb.amazonaws.com",
    "requestParameters":
    {
        "encryptionContext":
        {
            "aws:dynamodb:tableName": "Books",
            "aws:dynamodb:subscriberId": "111122223333"
        }
    },
    "responseElements": null,
    "requestID": "11cab293-11a6-11e8-8386-13160d3e5db5",
    "eventID": "b7d16574-e887-4b5b-a064-bf92f8ec9ad3",
    "readOnly": true,
    "resources": [
        {
            "ARN": "arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab",
            "accountId": "111122223333",
            "type": "AWS::KMS::Key"
        }
    ],
    "eventType": "AwsApiCall",
    "recipientAccountId": "111122223333"
}
```

CreateGrant

When you use a customer managed key or an AWS managed key to protect your DynamoDB table, DynamoDB uses grants to allow the service to perform continuous data protection and maintenance and durability tasks. These grants are not required on AWS owned key.

The grants that DynamoDB creates are specific to a table. The principal in the CreateGrant request is the user who created the table.

The event that records the `CreateGrant` operation is similar to the following example event. The parameters include the Amazon Resource Name (ARN) of the KMS key for the table, the grantee principal and retiring principal (the DynamoDB service), and the operations that the grant covers. It also includes a constraint that requires all encryption operation use the specified encryption context.

```
{
    "eventVersion": "1.05",
    "userIdentity":
    {
        "type": "AssumedRole",
        "principalId": "AROAIGDTESTANDEXAMPLE:user01",
        "arn": "arn:aws:sts::111122223333:assumed-role/Admin/user01",
        "accountId": "111122223333",
        "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
        "sessionContext": {
            "attributes": {
                "mfaAuthenticated": "false",
                "creationDate": "2018-02-14T00:12:02Z"
            },
            "sessionIssuer": {
                "type": "Role",
                "principalId": "AROAIGDTESTANDEXAMPLE",
                "arn": "arn:aws:iam::111122223333:role/Admin",
                "accountId": "111122223333",
                "userName": "Admin"
            }
        },
        "invokedBy": "dynamodb.amazonaws.com"
    },
    "eventTime": "2018-02-14T00:15:15Z",
    "eventSource": "kms.amazonaws.com",
    "eventName": "CreateGrant",
    "awsRegion": "us-west-2",
```

```
    "sourceIPAddress": "dynamodb.amazonaws.com",
    "userAgent": "dynamodb.amazonaws.com",
    "requestParameters": {
        "keyId": "1234abcd-12ab-34cd-56ef-1234567890ab",
        "retiringPrincipal": "dynamodb.us-west-2.amazonaws.com",
        "constraints": {
            "encryptionContextSubset": {
                "aws:dynamodb:tableName": "Books",
                "aws:dynamodb:subscriberId": "111122223333"
            }
        },
        "granteePrincipal": "dynamodb.us-west-2.amazonaws.com",
        "operations": [
            "DescribeKey",
            "GenerateDataKey",
            "Decrypt",
            "Encrypt",
            "ReEncryptFrom",
            "ReEncryptTo",
            "RetireGrant"
        ]
    },
    "responseElements": {
        "grantId":
 "5c5cd4a3d68e65e77795f5ccc2516dff057308172b0cd107c85b5215c6e48bde"
    },
    "requestID": "2192b82a-111c-11e8-a528-f398979205d8",
    "eventID": "a03d65c3-9fee-4111-9816-8bf96b73df01",
    "readOnly": false,
    "resources": [
        {
            "ARN": "arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab",
            "accountId": "111122223333",
            "type": "AWS::KMS::Key"
        }
    ],
    "eventType": "AwsApiCall",
    "recipientAccountId": "111122223333"
}
```

Managing encrypted tables in DynamoDB

You can use the AWS Management Console or the AWS Command Line Interface (AWS CLI) to specify the encryption key on new tables and update the encryption keys on existing tables in Amazon DynamoDB.

Topics

- Specifying the encryption key for a new table
- Updating an encryption key

Specifying the encryption key for a new table

Follow these steps to specify the encryption key on a new table using the Amazon DynamoDB console or the AWS CLI.

Creating an encrypted table (console)

1. Sign in to the AWS Management Console and open the DynamoDB console at https://console.aws.amazon.com/dynamodb/.
2. In the navigation pane on the left side of the console, choose **Tables**.
3. Choose **Create Table**. For the **Table name**, enter **`Music`**. For the primary key, enter **`Artist`**, and for the sort key, enter **`SongTitle`**, both as strings.
4. In **Settings**, make sure that **Customize settings** is selected.

   > **Note**
   >
   > If **Use default settings** is selected, tables are encrypted at rest with the AWS owned key at no additional cost.

5. Under **Encryption at rest**, choose an encryption type - AWS owned key, AWS managed key, or customer managed key.

   - **Owned by Amazon DynamoDB.** AWS owned key, specifically owned and managed by DynamoDB. You are not charged an additional fee for using this key.
   - **AWS managed key**. Key alias: `aws/dynamodb`. The key is stored in your account and is managed by AWS Key Management Service (AWS KMS). AWS KMS charges apply.

- **Stored in your account, and owned and managed by you.** Customer managed key. The key is stored in your account and is managed by AWS Key Management Service (AWS KMS). AWS KMS charges apply.

> **Note**
>
> If you select to own and manage your own key, make sure the KMS Key Policy is appropriately set. For more information including examples, see Key policy for a customer managed key.

6. Choose **Create table** to create the encrypted table. To confirm the encryption type, select the table details on the **Overview** tab and review the **Additional details** section.

Creating an encrypted table (AWS CLI)

Use the AWS CLI to create a table with the default AWS owned key, the AWS managed key, or a customer managed key for Amazon DynamoDB.

To create an encrypted table with the default AWS owned key

- Create the encrypted `Music` table as follows.

```
aws dynamodb create-table \
  --table-name Music \
  --attribute-definitions \
      AttributeName=Artist,AttributeType=S \
      AttributeName=SongTitle,AttributeType=S \
  --key-schema \
      AttributeName=Artist,KeyType=HASH \
      AttributeName=SongTitle,KeyType=RANGE \
  --provisioned-throughput \
      ReadCapacityUnits=10,WriteCapacityUnits=5
```

> **Note**
>
> This table is now encrypted using the default AWS owned key in the DynamoDB service account.

To create an encrypted table with the AWS managed key for DynamoDB

- Create the encrypted `Music` table as follows.

```
aws dynamodb create-table \
  --table-name Music \
  --attribute-definitions \
      AttributeName=Artist,AttributeType=S \
      AttributeName=SongTitle,AttributeType=S \
  --key-schema \
      AttributeName=Artist,KeyType=HASH \
      AttributeName=SongTitle,KeyType=RANGE \
  --provisioned-throughput \
      ReadCapacityUnits=10,WriteCapacityUnits=5 \
  --sse-specification Enabled=true,SSEType=KMS
```

The `SSEDescription` status of the table description is set to `ENABLED` and the `SSEType` is `KMS`.

```
"SSEDescription": {
  "SSEType": "KMS",
  "Status": "ENABLED",
  "KMSMasterKeyArn": "arn:aws:kms:us-east-1:123456789012:key/abcd1234-abcd-1234-
a123-ab1234a1b234",
}
```

To create an encrypted table with a customer managed key for DynamoDB

- Create the encrypted `Music` table as follows.

```
aws dynamodb create-table \
  --table-name Music \
  --attribute-definitions \
      AttributeName=Artist,AttributeType=S \
      AttributeName=SongTitle,AttributeType=S \
  --key-schema \
      AttributeName=Artist,KeyType=HASH \
      AttributeName=SongTitle,KeyType=RANGE \
  --provisioned-throughput \
      ReadCapacityUnits=10,WriteCapacityUnits=5 \
```

```
  --sse-specification Enabled=true,SSEType=KMS,KMSMasterKeyId=abcd1234-abcd-1234-
a123-ab1234a1b234
```

> **Note**
>
> For the `KMSMasterKeyId`, you can use a key ID, key ARN, or key alias. If you use a key alias (for example, `alias/my-key`), DynamoDB resolves the alias and associates the underlying AWS KMS key with the table. In the table description, the `KMSMasterKeyArn` will always show the key ARN of the resolved key, not the alias. For more information about key identifiers, see Key identifiers (KeyId) in the *AWS Key Management Service Developer Guide*.

The `SSEDescription` status of the table description is set to `ENABLED` and the `SSEType` is `KMS`.

```
"SSEDescription": {
  "SSEType": "KMS",
  "Status": "ENABLED",
  "KMSMasterKeyArn": "arn:aws:kms:us-east-1:123456789012:key/abcd1234-abcd-1234-
a123-ab1234a1b234",
}
```

Updating an encryption key

You can also use the DynamoDB console or the AWS CLI to update the encryption keys of an existing table between an AWS owned key, AWS managed key, and customer managed key at any time.

Updating an encryption key (console)

1. Sign in to the AWS Management Console and open the DynamoDB console at https://console.aws.amazon.com/dynamodb/.
2. In the navigation pane on the left side of the console, choose **Tables**.
3. Choose the table that you want to update.
4. Select the **Actions** dropdown, and then select the **Update settings** option.
5. Go to the **Additional settings** tab.

6. Under **Encryption**, choose **Manage encryption**.
7. Choose an encryption type:

   - **Owned by Amazon DynamoDB.** The AWS KMS key is owned and managed by DynamoDB. You are not charged an additional fee for using this key.
   - **AWS managed key** Key alias: `aws/dynamodb`. The key is stored in your account and is managed by AWS Key Management Service. (AWS KMS). AWS KMS charges apply.
   - **Stored in your account, and owned and managed by you.** The key is stored in your account and is managed by AWS Key Management Service. (AWS KMS). AWS KMS charges apply.

   > **Note**
   >
   > If you select to own and manage your own key, make sure the KMS Key Policy is appropriately set. For more information see Key policy for a customer managed key.

   Then choose **Save** to update the encrypted table. To confirm the encryption type, check the table details under the **Overview** tab.

Updating an encryption key (AWS CLI)

The following examples show how to update an encrypted table using the AWS CLI.

To update an encrypted table with the default AWS owned key

- Update the encrypted `Music` table, as in the following example.

```
aws dynamodb update-table \
  --table-name Music \
  --sse-specification Enabled=false
```

> **Note**
>
> This table is now encrypted using the default AWS owned key in the DynamoDB service account.

To update an encrypted table with the AWS managed key for DynamoDB

- Update the encrypted `Music` table, as in the following example.

```
aws dynamodb update-table \
  --table-name Music \
  --sse-specification Enabled=true
```

The `SSEDescription` status of the table description is set to `ENABLED` and the `SSEType` is `KMS`.

```
"SSEDescription": {
  "SSEType": "KMS",
  "Status": "ENABLED",
  "KMSMasterKeyArn": "arn:aws:kms:us-east-1:123456789012:key/abcd1234-abcd-1234-
a123-ab1234a1b234",
}
```

To update an encrypted table with a customer managed key for DynamoDB

- Update the encrypted `Music` table, as in the following example.

```
aws dynamodb update-table \
  --table-name Music \
  --sse-specification Enabled=true,SSEType=KMS,KMSMasterKeyId=abcd1234-abcd-1234-
a123-ab1234a1b234
```

> **Note**
>
> For the `KMSMasterKeyId`, you can use a key ID, key ARN, or key alias. If you use a key alias (for example, `alias/my-key`), DynamoDB resolves the alias and associates the underlying AWS KMS key with the table. In the table description, the `KMSMasterKeyArn` will always show the key ARN of the resolved key, not the alias.

The `SSEDescription` status of the table description is set to `ENABLED` and the `SSEType` is `KMS`.

```
"SSEDescription": {
  "SSEType": "KMS",
  "Status": "ENABLED",
  "KMSMasterKeyArn": "arn:aws:kms:us-east-1:123456789012:key/abcd1234-abcd-1234-
a123-ab1234a1b234",
}
```

Securing DynamoDB connections using VPC endpoints and IAM policies"

Connections are protected both between Amazon DynamoDB and on-premises applications and between DynamoDB and other AWS resources within the same AWS Region.

Required policy for endpoints

Amazon DynamoDB provides a DescribeEndpoints API that enables you to enumerate regional endpoint information. For requests to the public DynamoDB endpoints, the API responds regardless of the configured DynamoDB IAM policy, even if there is an explicit or implicit deny in the IAM or VPC endpoint policy. This is because DynamoDB intentionally skips authorization for the `DescribeEndpoints` API.

For requests from a VPC endpoint, both the IAM and Virtual Private Cloud (VPC) endpoint policies must authorize the `DescribeEndpoints` API call for the requesting Identity and Access Management (IAM) principal(s) using the IAM `dynamodb:DescribeEndpoints` action. Otherwise, access to the `DescribeEndpoints` API will be denied.

The following is an example of an endpoints policy.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
                "AWS": "arn:aws:iam::111122223333:root"
```

```
            },
            "Action": "dynamodb:DescribeEndpoints",
            "Resource": "*"
        }
    ]
}
```

Traffic between service and on-premises clients and applications

You have two connectivity options between your private network and AWS:

- An AWS Site-to-Site VPN connection. For more information, see What is AWS Site-to-Site VPN? in the *AWS Site-to-Site VPN User Guide*.
- An Direct Connect connection. For more information, see What is Direct Connect? in the *Direct Connect User Guide*.

Access to DynamoDB via the network is through AWS published APIs. Clients must support Transport Layer Security (TLS) 1.2. We recommend TLS 1.3. Clients must also support cipher suites with Perfect Forward Secrecy (PFS), such as Ephemeral Diffie-Hellman (DHE) or Elliptic Curve Diffie-Hellman Ephemeral (ECDHE). Most modern systems such as Java 7 and later support these modes. Additionally, you must sign requests using an access key ID and a secret access key that are associated with an IAM principal, or you can use the AWS Security Token Service (STS) to generate temporary security credentials to sign requests.

Traffic between AWS resources in the same Region

An Amazon Virtual Private Cloud (Amazon VPC) endpoint for DynamoDB is a logical entity within a VPC that allows connectivity only to DynamoDB. The Amazon VPC routes requests to DynamoDB and routes responses back to the VPC. For more information, see VPC endpoints in the *Amazon VPC User Guide*. For example policies that you can use to control access from VPC endpoints, see Using IAM policies to control access to DynamoDB.

> **Note**
>
> Amazon VPC endpoints are not accessible via AWS Site-to-Site VPN or Direct Connect.

AWS Identity and Access Management (IAM) and DynamoDB

AWS Identity and Access Management is an AWS service that helps an administrator securely control access to AWS resources. Administrators control who can be authenticated (signed in) and authorized (have permissions) to use Amazon DynamoDB and DynamoDB Accelerator resources. You can use IAM to manage access permissions and implement security policies for both Amazon DynamoDB and DynamoDB Accelerator. IAM is an AWS service that you can use with no additional charge.

Topics

- Identity and Access Management for Amazon DynamoDB
- Using IAM policy conditions for fine-grained access control

Identity and Access Management for Amazon DynamoDB

AWS Identity and Access Management (IAM) is an AWS service that helps an administrator securely control access to AWS resources. IAM administrators control who can be *authenticated* (signed in) and *authorized* (have permissions) to use DynamoDB resources. IAM is an AWS service that you can use with no additional charge.

Topics

- Audience
- Authenticating with identities
- Managing access using policies
- How Amazon DynamoDB works with IAM
- Identity-based policy examples for Amazon DynamoDB
- Troubleshooting Amazon DynamoDB identity and access
- IAM policy to prevent the purchase of DynamoDB reserved capacity

Audience

How you use AWS Identity and Access Management (IAM) differs based on your role:

- **Service user** - request permissions from your administrator if you cannot access features (see Troubleshooting Amazon DynamoDB identity and access)
- **Service administrator** - determine user access and submit permission requests (see How Amazon DynamoDB works with IAM)
- **IAM administrator** - write policies to manage access (see Identity-based policy examples for Amazon DynamoDB)

Authenticating with identities

Authentication is how you sign in to AWS using your identity credentials. You must be authenticated as the AWS account root user, an IAM user, or by assuming an IAM role.

You can sign in as a federated identity using credentials from an identity source like AWS IAM Identity Center (IAM Identity Center), single sign-on authentication, or Google/Facebook credentials. For more information about signing in, see How to sign in to your AWS account in the *AWS Sign-In User Guide*.

For programmatic access, AWS provides an SDK and CLI to cryptographically sign requests. For more information, see AWS Signature Version 4 for API requests in the *IAM User Guide*.

AWS account root user

When you create an AWS account, you begin with one sign-in identity called the AWS account *root user* that has complete access to all AWS services and resources. We strongly recommend that you don't use the root user for everyday tasks. For tasks that require root user credentials, see Tasks that require root user credentials in the *IAM User Guide*.

Federated identity

As a best practice, require human users to use federation with an identity provider to access AWS services using temporary credentials.

A *federated identity* is a user from your enterprise directory, web identity provider, or Directory Service that accesses AWS services using credentials from an identity source. Federated identities assume roles that provide temporary credentials.

For centralized access management, we recommend AWS IAM Identity Center. For more information, see What is IAM Identity Center? in the *AWS IAM Identity Center User Guide*.

IAM users and groups

An *IAM user* is an identity with specific permissions for a single person or application. We recommend using temporary credentials instead of IAM users with long-term credentials. For more information, see Require human users to use federation with an identity provider to access AWS using temporary credentials in the *IAM User Guide*.

An *IAM group* specifies a collection of IAM users and makes permissions easier to manage for large sets of users. For more information, see Use cases for IAM users in the *IAM User Guide*.

IAM roles

An *IAM role* is an identity with specific permissions that provides temporary credentials. You can assume a role by switching from a user to an IAM role (console) or by calling an AWS CLI or AWS API operation. For more information, see Methods to assume a role in the *IAM User Guide*.

IAM roles are useful for federated user access, temporary IAM user permissions, cross-account access, cross-service access, and applications running on Amazon EC2. For more information, see Cross account resource access in IAM in the *IAM User Guide*.

Managing access using policies

You control access in AWS by creating policies and attaching them to AWS identities or resources. A policy defines permissions when associated with an identity or resource. AWS evaluates these policies when a principal makes a request. Most policies are stored in AWS as JSON documents. For more information about JSON policy documents, see Overview of JSON policies in the *IAM User Guide*.

Using policies, administrators specify who has access to what by defining which **principal** can perform **actions** on what **resources**, and under what **conditions**.

By default, users and roles have no permissions. An IAM administrator creates IAM policies and adds them to roles, which users can then assume. IAM policies define permissions regardless of the method used to perform the operation.

Identity-based policies

Identity-based policies are JSON permissions policy documents that you attach to an identity (user, group, or role). These policies control what actions identities can perform, on which resources, and under what conditions. To learn how to create an identity-based policy, see Define custom IAM permissions with customer managed policies in the *IAM User Guide*.

Identity-based policies can be *inline policies* (embedded directly into a single identity) or *managed policies* (standalone policies attached to multiple identities). To learn how to choose between managed and inline policies, see Choose between managed policies and inline policies in the *IAM User Guide*.

Resource-based policies

Resource-based policies are JSON policy documents that you attach to a resource. Examples include IAM *role trust policies* and Amazon S3 *bucket policies*. In services that support resource-based policies, service administrators can use them to control access to a specific resource. You must specify a principal in a resource-based policy.

Resource-based policies are inline policies that are located in that service. You can't use AWS managed policies from IAM in a resource-based policy.

Other policy types

AWS supports additional policy types that can set the maximum permissions granted by more common policy types:

- **Permissions boundaries** – Set the maximum permissions that an identity-based policy can grant to an IAM entity. For more information, see Permissions boundaries for IAM entities in the *IAM User Guide*.
- **Service control policies (SCPs)** – Specify the maximum permissions for an organization or organizational unit in AWS Organizations. For more information, see Service control policies in the *AWS Organizations User Guide*.
- **Resource control policies (RCPs)** – Set the maximum available permissions for resources in your accounts. For more information, see Resource control policies (RCPs) in the *AWS Organizations User Guide*.
- **Session policies** – Advanced policies passed as a parameter when creating a temporary session for a role or federated user. For more information, see Session policies in the *IAM User Guide*.

Multiple policy types

When multiple types of policies apply to a request, the resulting permissions are more complicated to understand. To learn how AWS determines whether to allow a request when multiple policy types are involved, see Policy evaluation logic in the *IAM User Guide*.

How Amazon DynamoDB works with IAM

Before you use IAM to manage access to DynamoDB, learn what IAM features are available to use with DynamoDB.

IAM feature	DynamoDB support
Identity-based policies	Yes
Resource-based policies	Yes
Policy actions	Yes
Policy resources	Yes
Policy condition keys	Yes
ACLs	No
ABAC (tags in policies)	Yes
Temporary credentials	Yes
Principal permissions	Yes
Service roles	Yes
Service-linked roles	Yes

To get a high-level view of how DynamoDB and other AWS services work with most IAM features, see AWS services that work with IAM in the *IAM User Guide*.

Identity-based policies for DynamoDB

Supports identity-based policies: Yes

Identity-based policies are JSON permissions policy documents that you can attach to an identity, such as an IAM user, group of users, or role. These policies control what actions users and roles can perform, on which resources, and under what conditions. To learn how to create an identity-based policy, see Define custom IAM permissions with customer managed policies in the *IAM User Guide*.

With IAM identity-based policies, you can specify allowed or denied actions and resources as well as the conditions under which actions are allowed or denied. To learn about all of the elements that you can use in a JSON policy, see IAM JSON policy elements reference in the *IAM User Guide*.

Identity-based policy examples for DynamoDB

To view examples of DynamoDB identity-based policies, see Identity-based policy examples for Amazon DynamoDB.

Resource-based policies within DynamoDB

Supports resource-based policies: Yes

Resource-based policies are JSON policy documents that you attach to a resource. Examples of resource-based policies are IAM *role trust policies* and Amazon S3 *bucket policies*. In services that support resource-based policies, service administrators can use them to control access to a specific resource. For the resource where the policy is attached, the policy defines what actions a specified principal can perform on that resource and under what conditions. You must specify a principal in a resource-based policy. Principals can include accounts, users, roles, federated users, or AWS services.

To enable cross-account access, you can specify an entire account or IAM entities in another account as the principal in a resource-based policy. For more information, see Cross account resource access in IAM in the *IAM User Guide*.

Policy actions for DynamoDB

Supports policy actions: Yes

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

The `Action` element of a JSON policy describes the actions that you can use to allow or deny access in a policy. Include actions in a policy to grant permissions to perform the associated operation.

To see a list of DynamoDB actions, see Actions defined by Amazon DynamoDB in the *Service Authorization Reference*.

Policy actions in DynamoDB use the following prefix before the action:

```
aws
```

To specify multiple actions in a single statement, separate them with commas.

```
"Action": [
      "aws:action1",
      "aws:action2"
         ]
```

To view examples of DynamoDB identity-based policies, see Identity-based policy examples for Amazon DynamoDB.

Policy resources for DynamoDB

Supports policy resources: Yes

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

The `Resource` JSON policy element specifies the object or objects to which the action applies. As a best practice, specify a resource using its Amazon Resource Name (ARN). For actions that don't support resource-level permissions, use a wildcard (*) to indicate that the statement applies to all resources.

```
"Resource": "*"
```

To see a list of DynamoDB resource types and their ARNs, see Resources defined by Amazon DynamoDB in the *Service Authorization Reference*. To learn with which actions you can specify the ARN of each resource, see Actions defined by Amazon DynamoDB.

To view examples of DynamoDB identity-based policies, see Identity-based policy examples for Amazon DynamoDB.

Policy condition keys for DynamoDB

Supports service-specific policy condition keys: Yes

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

The `Condition` element specifies when statements execute based on defined criteria. You can create conditional expressions that use condition operators, such as equals or less than, to match the condition in the policy with values in the request. To see all AWS global condition keys, see AWS global condition context keys in the *IAM User Guide*.

To see a list of DynamoDB condition keys, see Condition keys for Amazon DynamoDB in the *Service Authorization Reference*. To learn with which actions and resources you can use a condition key, see Actions defined by Amazon DynamoDB.

To view examples of DynamoDB identity-based policies, see Identity-based policy examples for Amazon DynamoDB.

Access control lists (ACLs) in DynamoDB

Supports ACLs: No

Access control lists (ACLs) control which principals (account members, users, or roles) have permissions to access a resource. ACLs are similar to resource-based policies, although they do not use the JSON policy document format.

Attribute-based access control (ABAC) with DynamoDB

Supports ABAC (tags in policies): Yes

Attribute-based access control (ABAC) is an authorization strategy that defines permissions based on attributes called tags. You can attach tags to IAM entities and AWS resources, then design ABAC policies to allow operations when the principal's tag matches the tag on the resource.

To control access based on tags, you provide tag information in the condition element of a policy using the `aws:ResourceTag/key-name`, `aws:RequestTag/key-name`, or `aws:TagKeys` condition keys.

If a service supports all three condition keys for every resource type, then the value is **Yes** for the service. If a service supports all three condition keys for only some resource types, then the value is **Partial**.

For more information about ABAC, see Define permissions with ABAC authorization in the *IAM User Guide*. To view a tutorial with steps for setting up ABAC, see Use attribute-based access control (ABAC) in the *IAM User Guide*.

Using Temporary credentials with DynamoDB

Supports temporary credentials: Yes

Temporary credentials provide short-term access to AWS resources and are automatically created when you use federation or switch roles. AWS recommends that you dynamically generate temporary credentials instead of using long-term access keys. For more information, see Temporary security credentials in IAM and AWS services that work with IAM in the *IAM User Guide*.

Cross-service principal permissions for DynamoDB

Supports forward access sessions (FAS): Yes

Forward access sessions (FAS) use the permissions of the principal calling an AWS service, combined with the requesting AWS service to make requests to downstream services. For policy details when making FAS requests, see Forward access sessions.

Service roles for DynamoDB

Supports service roles: Yes

A service role is an IAM role that a service assumes to perform actions on your behalf. An IAM administrator can create, modify, and delete a service role from within IAM. For more information, see Create a role to delegate permissions to an AWS service in the *IAM User Guide*.

> **Warning**
>
> Changing the permissions for a service role might break DynamoDB functionality. Edit service roles only when DynamoDB provides guidance to do so.

Service-linked roles for DynamoDB

Supports service-linked roles: Yes

A service-linked role is a type of service role that is linked to an AWS service. The service can assume the role to perform an action on your behalf. Service-linked roles appear in your AWS account and are owned by the service. An IAM administrator can view, but not edit the permissions for service-linked roles.

For details about creating or managing service-linked roles, see AWS services that work with IAM. Find a service in the table that includes a `Yes` in the **Service-linked role** column. Choose the **Yes** link to view the service-linked role documentation for that service.

Supported service-linked roles in DynamoDB

The following service-linked roles are supported in DynamoDB.

- DynamoDB uses the service-linked role **AWSServiceRoleForDynamoDBReplication** for global tables replication across AWS Regions. See DynamoDB global tables security for more information about the **AWSServiceRoleForDynamoDBReplication** service-linked role.
- DynamoDB Accelerator (DAX) uses the service-linked role **AWSServiceRoleForDAX** for configuring and maintaining a DAX cluster. See the section called "Using service-linked roles for DAX" for more information about the **AWSServiceRoleForDAX** service-linked role.

In addition to these DynamoDB service-linked roles, DynamoDB uses the Application Auto Scaling service for automatically managing throughput settings on provisioned capacity mode tables. The Application Auto Scaling service uses the service-linked role **AWSServiceRoleForApplicationAutoScaling_DynamoDBTable** to manage throughput settings on DynamoDB tables that have auto scaling enabled. See Service-linked roles for Application Auto Scaling for more information.

Identity-based policy examples for Amazon DynamoDB

By default, users and roles don't have permission to create or modify DynamoDB resources. To grant users permission to perform actions on the resources that they need, an IAM administrator can create IAM policies.

To learn how to create an IAM identity-based policy by using these example JSON policy documents, see Create IAM policies (console) in the *IAM User Guide*.

For details about actions and resource types defined by DynamoDB, including the format of the ARNs for each of the resource types, see Actions, resources, and condition keys for Amazon DynamoDB in the *Service Authorization Reference*.

Topics

- Policy best practices
- Using the DynamoDB console
- Allow users to view their own permissions
- Using identity-based policies with Amazon DynamoDB

Policy best practices

Identity-based policies determine whether someone can create, access, or delete DynamoDB resources in your account. These actions can incur costs for your AWS account. When you create or edit identity-based policies, follow these guidelines and recommendations:

- **Get started with AWS managed policies and move toward least-privilege permissions** – To get started granting permissions to your users and workloads, use the *AWS managed policies* that grant permissions for many common use cases. They are available in your AWS account. We recommend that you reduce permissions further by defining AWS customer managed policies that are specific to your use cases. For more information, see AWS managed policies or AWS managed policies for job functions in the *IAM User Guide*.
- **Apply least-privilege permissions** – When you set permissions with IAM policies, grant only the permissions required to perform a task. You do this by defining the actions that can be taken on specific resources under specific conditions, also known as *least-privilege permissions*. For more information about using IAM to apply permissions, see Policies and permissions in IAM in the *IAM User Guide*.
- **Use conditions in IAM policies to further restrict access** – You can add a condition to your policies to limit access to actions and resources. For example, you can write a policy condition to specify that all requests must be sent using SSL. You can also use conditions to grant access to service actions if they are used through a specific AWS service, such as CloudFormation. For more information, see IAM JSON policy elements: Condition in the *IAM User Guide*.
- **Use IAM Access Analyzer to validate your IAM policies to ensure secure and functional permissions** – IAM Access Analyzer validates new and existing policies so that the policies adhere to the IAM policy language (JSON) and IAM best practices. IAM Access Analyzer provides more than 100 policy checks and actionable recommendations to help you author secure and functional policies. For more information, see Validate policies with IAM Access Analyzer in the *IAM User Guide*.
- **Require multi-factor authentication (MFA)** – If you have a scenario that requires IAM users or a root user in your AWS account, turn on MFA for additional security. To require MFA when API operations are called, add MFA conditions to your policies. For more information, see Secure API access with MFA in the *IAM User Guide*.

For more information about best practices in IAM, see Security best practices in IAM in the *IAM User Guide*.

Using the DynamoDB console

To access the Amazon DynamoDB console, you must have a minimum set of permissions. These permissions must allow you to list and view details about the DynamoDB resources in your AWS account. If you create an identity-based policy that is more restrictive than the minimum required permissions, the console won't function as intended for entities (users or roles) with that policy.

You don't need to allow minimum console permissions for users that are making calls only to the AWS CLI or the AWS API. Instead, allow access to only the actions that match the API operation that they're trying to perform.

To ensure that users and roles can still use the DynamoDB console, also attach the DynamoDB `ConsoleAccess` or `ReadOnly` AWS managed policy to the entities. For more information, see Adding permissions to a user in the *IAM User Guide*.

Allow users to view their own permissions

This example shows how you might create a policy that allows IAM users to view the inline and managed policies that are attached to their user identity. This policy includes permissions to complete this action on the console or programmatically using the AWS CLI or AWS API.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "ViewOwnUserInfo",
            "Effect": "Allow",
            "Action": [
                "iam:GetUserPolicy",
                "iam:ListGroupsForUser",
                "iam:ListAttachedUserPolicies",
                "iam:ListUserPolicies",
                "iam:GetUser"
            ],
            "Resource": ["arn:aws:iam::*:user/${aws:username}"]
        },
        {
            "Sid": "NavigateInConsole",
            "Effect": "Allow",
            "Action": [
                "iam:GetGroupPolicy",
                "iam:GetPolicyVersion",
                "iam:GetPolicy",
```

```
                "iam:ListAttachedGroupPolicies",
                "iam:ListGroupPolicies",
                "iam:ListPolicyVersions",
                "iam:ListPolicies",
                "iam:ListUsers"
            ],
            "Resource": "*"
        }
    ]
}
```

Using identity-based policies with Amazon DynamoDB

This topic covers using identity-based AWS Identity and Access Management (IAM) policies with Amazon DynamoDB and provides examples. The examples show how an account administrator can attach permissions policies to IAM identities (users, groups, and roles) and thereby grant permissions to perform operations on Amazon DynamoDB resources.

The sections in this topic cover the following:

- IAM permissions required to use the Amazon DynamoDB console
- AWS managed (predefined) IAM policies for Amazon DynamoDB
- Customer managed policy examples

The following is an example of a permissions policy.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "DescribeQueryScanBooksTable",
            "Effect": "Allow",
            "Action": [
                "dynamodb:DescribeTable",
                "dynamodb:Query",
                "dynamodb:Scan"
            ],
            "Resource": "arn:aws:dynamodb:us-west-2:111122223333:table/Books"
```

```
        }
    ]
}
```

The preceding policy has one statement that grants permissions for three DynamoDB actions (`dynamodb:DescribeTable`, `dynamodb:Query`, and `dynamodb:Scan`) on a table in the `us-west-2` AWS Region, which is owned by the AWS account specified by *`account-id`*. The *Amazon Resource Name (ARN)* in the `Resource` value specifies the table that the permissions apply to.

IAM permissions required to use the Amazon DynamoDB console

To work with the DynamoDB console, a user must have a minimum set of permissions that allow the user to work with their AWS account's DynamoDB resources. In addition to these DynamoDB permissions, the console requires permissions:

- Amazon CloudWatch permissions to display metrics and graphs.
- AWS Data Pipeline permissions to export and import DynamoDB data.
- AWS Identity and Access Management permissions to access roles necessary for exports and imports.
- Amazon Simple Notification Service permissions to notify you whenever a CloudWatch alarm is triggered.
- AWS Lambda permissions to process DynamoDB Streams records.

If you create an IAM policy that is more restrictive than the minimum required permissions, the console won't function as intended for users with that IAM policy. To ensure that those users can still use the DynamoDB console, also attach the `AmazonDynamoDBReadOnlyAccess` AWS managed policy to the user, as described in AWS managed (predefined) IAM policies for Amazon DynamoDB.

You don't need to allow minimum console permissions for users who are making calls only to the AWS CLI or the Amazon DynamoDB API.

> **Note**
>
> If you refer to a VPC endpoint, you will also need to authorize the DescribeEndpoints API call for the requesting IAM principal(s) with the IAM action (dynamodb:DescribeEndpoints). For more information see Required policy for endpoints.

AWS managed (predefined) IAM policies for Amazon DynamoDB

AWS addresses some common use cases by providing standalone IAM policies that are created and administered by AWS. These AWS managed policies grant necessary permissions for common use cases so that you can avoid having to investigate which permissions are needed. For more information, see AWS Managed Policies in the *IAM User Guide*.

The following AWS managed policies, which you can attach to users in your account, are specific to DynamoDB and are grouped by use-case scenario:

- **AmazonDynamoDBReadOnlyAccess** – Grants read-only access to DynamoDB resources through the AWS Management Console.
- **AmazonDynamoDBFullAccess** – Grants full access to DynamoDB resources through the AWS Management Console.

You can review these AWS managed permissions policies by signing in to the IAM console and searching for specific policies there.

> **Important**
>
> The best practice is to create custom IAM policies that grant least-privilege to the users, roles, or groups that require them.

Customer managed policy examples

In this section, you can find policy examples that grant permissions for various DynamoDB actions. These policies work when you use AWS SDKs or the AWS CLI. When you use the console, you need to grant additional permissions that are specific to the console. For more information, see IAM permissions required to use the Amazon DynamoDB console.

> **Note**
>
> All of the following policy examples use one of the AWS Regions and contain fictitious account IDs and table names.

Examples:

- IAM policy to grant permissions to all DynamoDB actions on a table
- IAM policy to grant read-only permissions on items in a DynamoDB table
- IAM policy to grant access to a specific DynamoDB table and its indexes
- IAM policy to read, write, update, and delete access on a DynamoDB table
- IAM policy to separate DynamoDB environments in the same AWS account
- IAM policy to prevent the purchase of DynamoDB reserved capacity
- IAM policy to grant read access for a DynamoDB stream only (not for the table)
- IAM policy to allow an AWS Lambda function to access DynamoDB stream records
- IAM policy for read and write access to a DynamoDB Accelerator (DAX) cluster

The *IAM User Guide*, includes three additional DynamoDB examples:

- Amazon DynamoDB: Allows Access to a Specific Table
- Amazon DynamoDB: Allows Access to Specific Columns
- Amazon DynamoDB: Allows Row-Level Access to DynamoDB Based on an Amazon Cognito ID

IAM policy to grant permissions to all DynamoDB actions on a table

The following policy grants permissions for *all* DynamoDB actions on a table called `Books`. The resource ARN specified in the `Resource` identifies a table in a specific AWS Region. If you replace the table name `Books` in the `Resource` ARN with a wildcard character (*), *all* DynamoDB actions are allowed on *all* tables in the account. Carefully consider the possible security implications before using a wildcard character on this or any IAM policy.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "AllAPIActionsOnBooks",
            "Effect": "Allow",
            "Action": "dynamodb:*",
            "Resource": "arn:aws:dynamodb:us-west-2:123456789012:table/Books"
        }
    ]
```

```
}
```

> **Note**
>
> This is an example of using a wildcard character (*) to allow *all* actions, including administration, data operations, monitoring, and purchase of DynamoDB reserved capacity. Instead, it is a best practice to explicitly specify each action to be granted and only what that user, role, or group needs.

IAM policy to grant read-only permissions on items in a DynamoDB table

The following permissions policy grants permissions for the `GetItem`, `BatchGetItem`, `Scan`, `Query`, and `ConditionCheckItem` DynamoDB actions only, and as a result, sets read-only access on the `Books` table.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "ReadOnlyAPIActionsOnBooks",
            "Effect": "Allow",
            "Action": [
                "dynamodb:GetItem",
                "dynamodb:BatchGetItem",
                "dynamodb:Scan",
                "dynamodb:Query",
                "dynamodb:ConditionCheckItem"
            ],
            "Resource": "arn:aws:dynamodb:us-west-2:123456789012:table/Books"
        }
    ]
}
```

IAM policy to grant access to a specific DynamoDB table and its indexes

The following policy grants permissions for data modification actions on a DynamoDB table called Books and all of that table's indexes. For more information about how indexes work, see Improving data access with secondary indexes in DynamoDB.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "AccessTableAllIndexesOnBooks",
            "Effect": "Allow",
            "Action": [
              "dynamodb:PutItem",
              "dynamodb:UpdateItem",
              "dynamodb:DeleteItem",
              "dynamodb:BatchWriteItem",
              "dynamodb:GetItem",
              "dynamodb:BatchGetItem",
              "dynamodb:Scan",
              "dynamodb:Query",
              "dynamodb:ConditionCheckItem"
            ],
            "Resource": [
                "arn:aws:dynamodb:us-west-2:123456789012:table/Books",
                "arn:aws:dynamodb:us-west-2:123456789012:table/Books/index/*"
            ]
        }
    ]
}
```

IAM policy to read, write, update, and delete access on a DynamoDB table

Use this policy if you need to allow your application to create, read, update, and delete data in Amazon DynamoDB tables, indexes, and streams. Substitute the AWS Region name, your account ID, and the table name or wildcard character (*) where appropriate.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "DynamoDBIndexAndStreamAccess",
            "Effect": "Allow",
            "Action": [
                "dynamodb:GetShardIterator",
                "dynamodb:Scan",
                "dynamodb:Query",
                "dynamodb:DescribeStream",
                "dynamodb:GetRecords",
                "dynamodb:ListStreams"
            ],
            "Resource": [
                "arn:aws:dynamodb:us-west-2:123456789012:table/Books/index/*",
                "arn:aws:dynamodb:us-west-2:123456789012:table/Books/stream/*"
            ]
        },
        {
            "Sid": "DynamoDBTableAccess",
            "Effect": "Allow",
            "Action": [
                "dynamodb:BatchGetItem",
                "dynamodb:BatchWriteItem",
                "dynamodb:ConditionCheckItem",
                "dynamodb:PutItem",
                "dynamodb:DescribeTable",
                "dynamodb:DeleteItem",
                "dynamodb:GetItem",
                "dynamodb:Scan",
                "dynamodb:Query",
                "dynamodb:UpdateItem"
            ],
            "Resource": "arn:aws:dynamodb:us-west-2:123456789012:table/Books"
        },
        {
            "Sid": "DynamoDBDescribeLimitsAccess",
            "Effect": "Allow",
            "Action": "dynamodb:DescribeLimits",
            "Resource": [
```

```
                "arn:aws:dynamodb:us-west-2:123456789012:table/Books",
                "arn:aws:dynamodb:us-west-2:123456789012:table/Books/index/*"
            ]
        }
    ]
}
```

To expand this policy to cover all DynamoDB tables in all AWS Regions for this account, use a wildcard (*) for the Region and table name. For example:

```
"Resource":[
               "arn:aws:dynamodb:*:123456789012:table/*",
               "arn:aws:dynamodb:*:123456789012:table/*/index/*"
               ]
```

IAM policy to separate DynamoDB environments in the same AWS account

Suppose that you have separate environments where each environment maintains its own version of a table named `ProductCatalog`. If you create two `ProductCatalog` tables in the same AWS account, work in one environment might affect the other environment because of the way that permissions are set up. For example, quotas on the number of concurrent control plane operations (such as `CreateTable`) are set at the AWS account level.

As a result, each action in one environment reduces the number of operations available in the other environment. There is also a risk that the code in one environment might accidentally access tables in the other environment.

> **Note**
>
> If you want to separate production and test workloads to help control an event's potential "blast radius," the best practice is to create separate AWS accounts for test and production workloads. For more information, see AWS Account Management and Separation.

Suppose further that you have two developers, Amit and Alice, who are testing the `ProductCatalog` table. Instead of each developer requiring a separate AWS account, your developers can share the same test AWS account. In this test account, you can create a copy of the same table for each developer to work on, such as `Alice_ProductCatalog` and

`Amit_ProductCatalog`. In this case, you can create users Alice and Amit in the AWS account that you created for the test environment. You then can grant permissions to these users to perform DynamoDB actions on the tables that they own.

To grant these IAM user permissions, you can do either of the following:

- Create a separate policy for each user, and then attach each policy to its user separately. For example, you can attach the following policy to user Alice to allow her access to DynamoDB actions on the `Alice_ProductCatalog` table:

  JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "AllAPIActionsOnAliceTable",
            "Effect": "Allow",
            "Action": [
              "dynamodb:DeleteItem",
              "dynamodb:DescribeContributorInsights",
              "dynamodb:RestoreTableToPointInTime",
              "dynamodb:ListTagsOfResource",
              "dynamodb:CreateTableReplica",
              "dynamodb:UpdateContributorInsights",
              "dynamodb:CreateBackup",
              "dynamodb:DeleteTable",
              "dynamodb:UpdateTableReplicaAutoScaling",
              "dynamodb:UpdateContinuousBackups",
              "dynamodb:TagResource",
              "dynamodb:DescribeTable",
              "dynamodb:GetItem",
              "dynamodb:DescribeContinuousBackups",
              "dynamodb:BatchGetItem",
              "dynamodb:UpdateTimeToLive",
              "dynamodb:BatchWriteItem",
              "dynamodb:ConditionCheckItem",
              "dynamodb:UntagResource",
              "dynamodb:PutItem",
              "dynamodb:Scan",
              "dynamodb:Query",
              "dynamodb:UpdateItem",
              "dynamodb:DeleteTableReplica",
```

```
                "dynamodb:DescribeTimeToLive",
                "dynamodb:RestoreTableFromBackup",
                "dynamodb:UpdateTable",
                "dynamodb:DescribeTableReplicaAutoScaling",
                "dynamodb:GetShardIterator",
                "dynamodb:DescribeStream",
                "dynamodb:GetRecords",
                "dynamodb:DescribeLimits",
                "dynamodb:ListStreams"
            ],
            "Resource": "arn:aws:dynamodb:us-west-2:123456789012:table/
Alice_ProductCatalog/*"
        }
    ]
}
```

Then, you can create a similar policy with a different resource (the `Amit_ProductCatalog` table) for user Amit.

- Instead of attaching policies to individual users, you can use IAM policy variables to write a single policy and attach it to a group. You need to create a group and, for this example, add both users Alice and Amit to the group. The following example grants permissions to perform all DynamoDB actions on the `${aws:username}_ProductCatalog` table. The policy variable `${aws:username}` is replaced by the requester's user name when the policy is evaluated. For example, if Alice sends a request to add an item, the action is allowed only if Alice is adding items to the `Alice_ProductCatalog` table.

  JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "ActionsOnUserSpecificTable",
            "Effect": "Allow",
            "Action": [
                "dynamodb:PutItem",
                "dynamodb:UpdateItem",
                "dynamodb:DeleteItem",
                "dynamodb:BatchWriteItem",
                "dynamodb:GetItem",
                "dynamodb:BatchGetItem",
                "dynamodb:Scan",
```

```
                "dynamodb:Query",
                "dynamodb:ConditionCheckItem"
            ],
            "Resource": "arn:aws:dynamodb:us-west-2:123456789012:table/
${aws:username}_ProductCatalog"
        },
        {
            "Sid": "AdditionalPrivileges",
            "Effect": "Allow",
            "Action": [
                "dynamodb:ListTables",
                "dynamodb:DescribeTable",
                "dynamodb:DescribeContributorInsights"
            ],
            "Resource": "arn:aws:dynamodb:us-west-2:123456789012:table/*"
        }
    ]
}
```

Note

When using IAM policy variables, you must explicitly specify the `2012-10-17` version of the IAM policy language in the policy. The default version of the IAM policy language (`2008-10-17`) does not support policy variables.

Instead of identifying a specific table as a resource as you normally would, you could use a wildcard character (*) to grant permissions on all tables where the table name is prefixed with the user that is making the request, as shown in the following example.

```
"Resource":"arn:aws:dynamodb:us-west-2:123456789012:table/${aws:username}_*"
```

IAM policy to prevent the purchase of DynamoDB reserved capacity

With Amazon DynamoDB reserved capacity, you pay a one-time, upfront fee and commit to paying for a minimum usage level at significant savings over a period of time. You can use the AWS Management Console to view and purchase reserved capacity. However, you might not want all of the users in your organization to be able to purchase reserved capacity. For more information about reserved capacity, see Amazon DynamoDB pricing.

DynamoDB provides the following API operations for controlling access to reserved capacity management:

- `dynamodb:DescribeReservedCapacity` – Returns the reserved capacity purchases that are currently in effect.
- `dynamodb:DescribeReservedCapacityOfferings` – Returns details about the reserved capacity plans that are currently offered by AWS.
- `dynamodb:PurchaseReservedCapacityOfferings` – Performs an actual purchase of reserved capacity.

The AWS Management Console uses these API actions to display reserved capacity information and make purchases. You cannot call these operations from an application program because they can be accessed only from the console. However, you can allow or deny access to these operations in an IAM permissions policy.

The following policy allows users to view reserved capacity purchases and offerings by using the AWS Management Console — but new purchases are denied.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "AllowReservedCapacityDescriptions",
            "Effect": "Allow",
            "Action": [
                "dynamodb:DescribeReservedCapacity",
                "dynamodb:DescribeReservedCapacityOfferings"
            ],
            "Resource": "arn:aws:dynamodb:us-west-2:123456789012:*"
        },
        {
            "Sid": "DenyReservedCapacityPurchases",
            "Effect": "Deny",
            "Action": "dynamodb:PurchaseReservedCapacityOfferings",
            "Resource": "arn:aws:dynamodb:us-west-2:123456789012:*"
        }
    ]
```

```
}
```

Note that this policy uses the wildcard character (*) to allow describe permissions for all, and to deny the purchase of DynamoDB reserved capacity for all.

IAM policy to grant read access for a DynamoDB stream only (not for the table)

When you enable DynamoDB Streams on a table, information is captured about every modification to items in the table. For more information, see Change data capture for DynamoDB Streams.

In some cases, you might want to prevent an application from reading data from a DynamoDB table, but still allow access to that table's streams. For example, you can configure AWS Lambda to poll a stream and invoke a Lambda function when item updates are detected, and then perform additional processing.

The following actions are available for controlling access to DynamoDB streams:

- `dynamodb:DescribeStream`
- `dynamodb:GetRecords`
- `dynamodb:GetShardIterator`
- `dynamodb:ListStreams`

The following example policy grants users permissions to access the streams of a table named `GameScores`. The wildcard character (*) in the ARN matches any stream associated with that table.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "AccessGameScoresStreamOnly",
            "Effect": "Allow",
            "Action": [
                "dynamodb:DescribeStream",
                "dynamodb:GetRecords",
                "dynamodb:GetShardIterator",
                "dynamodb:ListStreams"
```

```
            ],
            "Resource": "arn:aws:dynamodb:us-west-2:123456789012:table/
GameScores/stream/*"
        }
    ]
}
```

Note that this policy grants access to the `GameScores` table's streams, but not to the table itself.

IAM policy to allow an AWS Lambda function to access DynamoDB stream records

If you want certain actions to be performed based on events in a DynamoDB stream, you can write an AWS Lambda function that is triggered by these events. A Lambda function such as this needs permissions to read data from a DynamoDB stream. For more information about using Lambda with DynamoDB Streams, see DynamoDB Streams and AWS Lambda triggers.

To grant permissions to Lambda, use the permissions policy that is associated with the Lambda function's IAM role (also known as an execution role). Specify this policy when you create the Lambda function.

For example, you can associate the following permissions policy with an execution role to grant Lambda permissions to perform the DynamoDB Streams actions listed.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "APIAccessForDynamoDBStreams",
            "Effect": "Allow",
            "Action": [
                "dynamodb:GetRecords",
                "dynamodb:GetShardIterator",
                "dynamodb:DescribeStream",
                "dynamodb:ListStreams"
            ],
            "Resource": "arn:aws:dynamodb:us-west-2:123456789012:table/
GameScores/stream/*"
        }
    ]
```

```
}
```

For more information, see AWS Lambda permissions in the *AWS Lambda Developer Guide*.

IAM policy for read and write access to a DynamoDB Accelerator (DAX) cluster

The following policy allows read, write, update, and delete access to a DynamoDB Accelerator (DAX) cluster, but not to the associated DynamoDB table. To use this policy, substitute the AWS Region name, your account ID, and the name of your DAX cluster.

> **Note**
>
> This policy gives access to DAX cluster, but not to the associated DynamoDB table. Make sure that your DAX cluster has the correct policy to perform these same operations on the DynamoDB table on your behalf.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "AmazonDynamoDBDAXDataOperations",
            "Effect": "Allow",
            "Action": [
                "dax:GetItem",
                "dax:PutItem",
                "dax:ConditionCheckItem",
                "dax:BatchGetItem",
                "dax:BatchWriteItem",
                "dax:DeleteItem",
                "dax:Query",
                "dax:UpdateItem",
                "dax:Scan"
            ],
            "Resource": "arn:aws:dax:eu-west-1:123456789012:cache/MyDAXCluster"
        }
    ]
}
```

To expand this policy to cover DAX access for all AWS Regions for an account, use a wildcard character (*) for the Region name.

```
"Resource": "arn:aws:dax:*:123456789012:cache/MyDAXCluster"
```

Troubleshooting Amazon DynamoDB identity and access

Use the following information to help you diagnose and fix common issues that you might encounter when working with DynamoDB and IAM.

Topics

- I am not authorized to perform an action in DynamoDB
- I am not authorized to perform iam:PassRole
- I want to allow people outside of my AWS account to access my DynamoDB resources

I am not authorized to perform an action in DynamoDB

If the AWS Management Console tells you that you're not authorized to perform an action, then you must contact your administrator for assistance. Your administrator is the person that provided you with your user name and password.

The following example error occurs when the `mateojackson` user tries to use the console to view details about a fictional *`my-example-widget`* resource but does not have the fictional `aws:`*`GetWidget`* permissions.

```
User: arn:aws:iam::123456789012:user/mateojackson is not authorized to perform:
 aws:GetWidget on resource: my-example-widget
```

In this case, Mateo asks his administrator to update his policies to allow him to access the *`my-example-widget`* resource using the `aws:`*`GetWidget`* action.

I am not authorized to perform iam:PassRole

If you receive an error that you're not authorized to perform the `iam:PassRole` action, your policies must be updated to allow you to pass a role to DynamoDB.

Some AWS services allow you to pass an existing role to that service instead of creating a new service role or service-linked role. To do this, you must have permissions to pass the role to the service.

The following example error occurs when an IAM user named `marymajor` tries to use the console to perform an action in DynamoDB. However, the action requires the service to have permissions that are granted by a service role. Mary does not have permissions to pass the role to the service.

```
User: arn:aws:iam::123456789012:user/marymajor is not authorized to perform:
 iam:PassRole
```

In this case, Mary's policies must be updated to allow her to perform the `iam:PassRole` action.

If you need help, contact your AWS administrator. Your administrator is the person who provided you with your sign-in credentials.

I want to allow people outside of my AWS account to access my DynamoDB resources

You can create a role that users in other accounts or people outside of your organization can use to access your resources. You can specify who is trusted to assume the role. For services that support resource-based policies or access control lists (ACLs), you can use those policies to grant people access to your resources.

To learn more, consult the following:

- To learn whether DynamoDB supports these features, see How Amazon DynamoDB works with IAM.
- To learn how to provide access to your resources across AWS accounts that you own, see Providing access to an IAM user in another AWS account that you own in the *IAM User Guide*.
- To learn how to provide access to your resources to third-party AWS accounts, see Providing access to AWS accounts owned by third parties in the *IAM User Guide*.
- To learn how to provide access through identity federation, see Providing access to externally authenticated users (identity federation) in the *IAM User Guide*.
- To learn the difference between using roles and resource-based policies for cross-account access, see Cross account resource access in IAM in the *IAM User Guide*.

IAM policy to prevent the purchase of DynamoDB reserved capacity

With Amazon DynamoDB reserved capacity, you pay a one-time, upfront fee and commit to paying for a minimum usage level at significant savings over a period of time. You can use the AWS Management Console to view and purchase reserved capacity. However, you might not want all of the users in your organization to be able to purchase reserved capacity. For more information about reserved capacity, see Amazon DynamoDB pricing.

DynamoDB provides the following API operations for controlling access to reserved capacity management:

- `dynamodb:DescribeReservedCapacity` – Returns the reserved capacity purchases that are currently in effect.
- `dynamodb:DescribeReservedCapacityOfferings` – Returns details about the reserved capacity plans that are currently offered by AWS.
- `dynamodb:PurchaseReservedCapacityOfferings` – Performs an actual purchase of reserved capacity.

The AWS Management Console uses these API actions to display reserved capacity information and make purchases. You cannot call these operations from an application program because they can be accessed only from the console. However, you can allow or deny access to these operations in an IAM permissions policy.

The following policy allows users to view reserved capacity purchases and offerings by using the AWS Management Console — but new purchases are denied.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "AllowReservedCapacityDescriptions",
            "Effect": "Allow",
            "Action": [
                "dynamodb:DescribeReservedCapacity",
                "dynamodb:DescribeReservedCapacityOfferings"
            ],
```

```
            "Resource": "arn:aws:dynamodb:us-west-2:123456789012:*"
        },
        {
            "Sid": "DenyReservedCapacityPurchases",
            "Effect": "Deny",
            "Action": "dynamodb:PurchaseReservedCapacityOfferings",
            "Resource": "arn:aws:dynamodb:us-west-2:123456789012:*"
        }
    ]
}
```

Note that this policy uses the wildcard character (*) to allow describe permissions for all, and to deny the purchase of DynamoDB reserved capacity for all.

Using IAM policy conditions for fine-grained access control

When you grant permissions in DynamoDB, you can specify conditions that determine how a permissions policy takes effect.

Overview

In DynamoDB, you have the option to specify conditions when granting permissions using an IAM policy (see Identity and Access Management for Amazon DynamoDB). For example, you can:

- Grant permissions to allow users read-only access to certain items and attributes in a table or a secondary index.
- Grant permissions to allow users write-only access to certain attributes in a table, based upon the identity of that user.

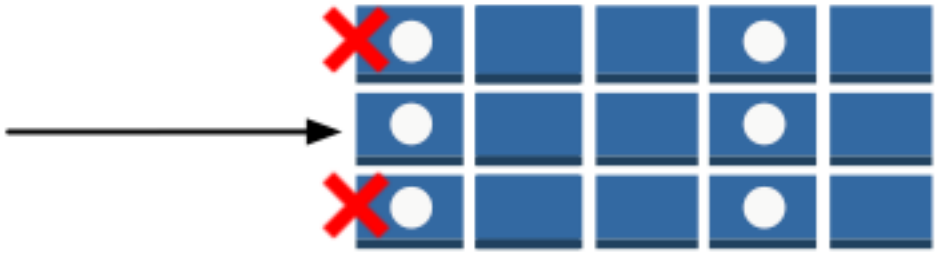
In DynamoDB, you can specify conditions in an IAM policy using condition keys, as illustrated in the use case in the following section.

Permissions use case

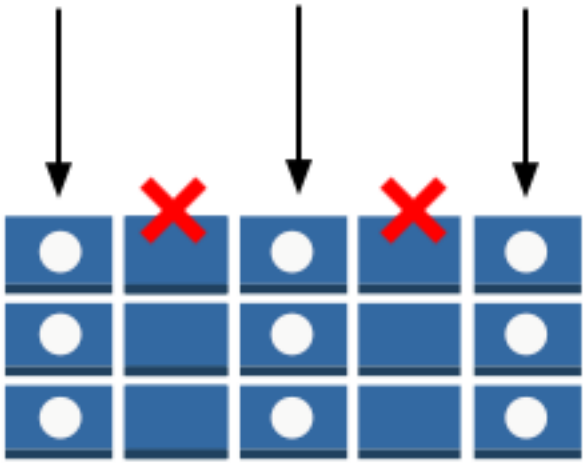
In addition to controlling access to DynamoDB API actions, you can also control access to individual data items and attributes. For example, you can do the following:

- Grant permissions on a table, but restrict access to specific items in that table based on certain primary key values. An example might be a social networking app for games, where all users'

saved game data is stored in a single table, but no users can access data items that they do not own, as shown in the following illustration:



- Hide information so that only a subset of attributes is visible to the user. An example might be an app that displays flight data for nearby airports, based on the user's location. Airline names, arrival and departure times, and flight numbers are all displayed. However, attributes such as pilot names or the number of passengers are hidden, as shown in the following illustration:



To implement this kind of fine-grained access control, you write an IAM permissions policy that specifies conditions for accessing security credentials and the associated permissions. You then apply the policy to users, groups, or roles that you create using the IAM console. Your IAM policy can restrict access to individual items in a table, access to the attributes in those items, or both at the same time.

You can optionally use web identity federation to control access by users who are authenticated by Login with Amazon, Facebook, or Google. For more information, see Using web identity federation.

You use the IAM `Condition` element to implement a fine-grained access control policy. By adding a `Condition` element to a permissions policy, you can allow or deny access to items and attributes in DynamoDB tables and indexes, based upon your particular business requirements.

The video below explains fine-grained access control in DynamoDB using IAM policy conditions.

Understanding Fine-Grained Access Control in DynamoDB

Fine-grained access control in DynamoDB allows you to create precise permissions boundaries at multiple levels:

1. **Item-level access control:** Restrict users to only access items that contain specific key values, typically matching their identity or permission scope.
2. **Attribute-level access control:** Limit which attributes (columns) users can view or modify, enabling you to protect sensitive information while allowing access to non-sensitive data within the same items.
3. **Operation-specific controls:** Apply different permission rules based on the type of operation being performed.

These controls are implemented through IAM policies using DynamoDB-specific condition keys.

Specifying conditions: Using condition keys

AWS provides a set of predefined condition keys (AWS-wide condition keys) for all AWS services that support IAM for access control. For example, you can use the `aws:SourceIp` condition key to check the requester's IP address before allowing an action to be performed. For more information and a list of the AWS-wide keys, see Available Keys for Conditions in the IAM User Guide.

The following are the DynamoDB service-specific condition keys that apply to DynamoDB.

`dynamodb:LeadingKeys`

Represents the first key attribute of a table—in other words, the partition key. The key name `LeadingKeys` is plural, even if the key is used with single-item actions. In addition, you must use the `ForAllValues` modifier when using `LeadingKeys` in a condition.

`dynamodb:Select`

Represents the `Select` parameter of a request. `Select` can be any of the following values:

- `ALL_ATTRIBUTES`

- `ALL_PROJECTED_ATTRIBUTES`
- `SPECIFIC_ATTRIBUTES`
- `COUNT`

> **Note**
>
> While often associated with Query and Scan operations, this condition key applies to all DynamoDB operations that return item attributes and is essential for controlling attribute access across all API actions. Using StringEqualsIfExists or similar constraints on this condition key will apply constraints on operations where this condition key applies, while ignoring it on operations where it does not apply.

`dynamodb:Attributes`

Represents a list of the *top-level* attributes accessed by a request. A top-level attribute is accessed by a request if it, or any nested attribute that it contains, is specified in the request parameters. For example, a `GetItem` request that specifies a `ProjectionExpression` of `"Name, Address.City"`, the `dynamodb:Attributes` list would include "Name" and "Address". If the `Attributes` parameter is enumerated in a fine grained access control policy then consider also restricting `ReturnValues` and `Select` parameters to ensure restricted access to specified attributes across multiple API actions like `GetItem`, `Query`, and `Scan`.

> **Note**
>
> This condition is evaluated only on the attributes specified in the request (such as in a ProjectionExpression), not on attributes in the response. If no ProjectionExpression is provided in the request, all attributes will be returned regardless of any attribute restrictions in the policy. See the section "Ensuring attribute-based restrictions are enforced" below for details on how to properly secure attribute access.

`dynamodb:ReturnValues`

Represents the `ReturnValues` parameter of a request. Depending on the API action, `ReturnValues` could be any of the following values:

- `ALL_OLD`

- UPDATED_OLD
- ALL_NEW
- UPDATED_NEW
- NONE

dynamodb:ReturnConsumedCapacity

Represents the ReturnConsumedCapacity parameter of a request. ReturnConsumedCapacity can be one of the following values:

- TOTAL
- NONE

dynamodb:FirstPartitionKeyValues

Represents the first key attribute of a table—in other words, the first partition key. The key name FirstPartitionKeyValues is plural, even if the key is used with single-item actions. In addition, you must use the ForAllValues modifier when using FirstPartitionKeyValues in a condition. FirstPartitionKeyValues and LeadingKeys can used exchangeable.

dynamodb:SecondPartitionKeyValues

Similar to dynamodb:FirstPartitionKeyValues. Represents resources' second partition key. The key name SecondPartitionKeyValues is plural, even if the key is used with single-item actions.

dynamodb:ThirdPartitionKeyValues

Similar to dynamodb:FirstPartitionKeyValues. Represents resources' third partition key. The key name ThirdPartitionKeyValues is plural, even if the key is used with single-item actions.

dynamodb:FourthPartitionKeyValues

Similar to dynamodb:FirstPartitionKeyValues. Represents resources' fourth partition key. The key name FourthPartitionKeyValues is plural, even if the key is used with single-item actions.

Ensuring attribute-based restrictions are enforced

When using attribute-based conditions to restrict access to specific attributes, it's important to understand how these conditions are evaluated:

- **Attribute conditions are evaluated only on attributes specified in the request**, not on attributes in the response.
- **For read operations without a ProjectionExpression** (GetItem, Query, Scan, etc.), all attributes will be returned regardless of attribute restrictions in your policy. To prevent this potential exposure of sensitive data, implement both attribute conditions (`dynamodb:Attributes`) and a condition requiring specific attributes must be requested (`dynamodb:Select`).
- **For write operations** (PutItem, UpdateItem, DeleteItem), the ReturnValues parameter can return complete items, potentially exposing restricted attributes even when the write operation itself complies with your policy. To prevent this exposure, implement both attribute conditions (`dynamodb:Attributes`) and restrictions on ReturnValues (`dynamodb:ReturnValues`) in your policy.

Limiting user access

Many IAM permissions policies allow users to access only those items in a table where the partition key value matches the user identifier. For example, the game app preceding limits access in this way so that users can only access game data that is associated with their user ID. The IAM substitution variables `${www.amazon.com:user_id}`, `${graph.facebook.com:id}`, and `${accounts.google.com:sub}` contain user identifiers for Login with Amazon, Facebook, and Google. To learn how an application logs in to one of these identity providers and obtains these identifiers, see Using web identity federation.

Important

Fine-grained access control isn't supported for restricting global tables replication. Applying policy conditions for fine-grained access control to DynamoDB service principals or service-linked roles used for global tables replication may interrupt replication within a global table.

Note

Each of the examples in the following section sets the `Effect` clause to `Allow` and specifies only the actions, resources, and parameters that are allowed. Access is permitted only to what is explicitly listed in the IAM policy.
In some cases, it is possible to rewrite these policies so that they are deny-based (that is, setting the `Effect` clause to `Deny` and inverting all of the logic in the policy). However,

we recommend that you avoid using deny-based policies with DynamoDB because they're difficult to write correctly, compared to allow-based policies. In addition, future changes to the DynamoDB API (or changes to existing API inputs) can render a deny-based policy ineffective.

Example policies: Using conditions for fine-grained access control

This section shows several policies for implementing fine-grained access control on DynamoDB tables and indexes.

Note

All examples use the us-west-2 Region and contain fictitious account IDs.

Example 1. Basic partition key-based access control with attribute restrictions

As an example, consider a mobile gaming app that lets players select from and play a variety of different games. The app uses a DynamoDB table named `GameScores` to keep track of high scores and other user data. Each item in the table is uniquely identified by a user ID and the name of the game that the user played. The `GameScores` table has a primary key consisting of a partition key (`UserId`) and sort key (`GameTitle`). Users only have access to game data associated with their user ID. A user who wants to play a game must belong to an IAM role named `GameRole`, which has a security policy attached to it.

To manage user permissions in this app, you could write a permissions policy such as the following:

JSON

```
{
   "Version":"2012-10-17",
   "Statement":[
      {
         "Sid":"AllowAccessToOnlyItemsMatchingUserID",
         "Effect":"Allow",
         "Action":[
            "dynamodb:GetItem",
            "dynamodb:BatchGetItem",
```

```
            "dynamodb:Query",
            "dynamodb:PutItem",
            "dynamodb:UpdateItem",
            "dynamodb:DeleteItem",
            "dynamodb:BatchWriteItem"
         ],
         "Resource":[
            "arn:aws:dynamodb:us-west-2:123456789012:table/GameScores"
         ],
         "Condition":{
            "ForAllValues:StringEquals":{
               "dynamodb:LeadingKeys":[
                  "${www.amazon.com:user_id}"
               ],
               "dynamodb:Attributes":[
                  "UserId",
                  "GameTitle",
                  "Wins",
                  "Losses",
                  "TopScore",
                  "TopScoreDateTime"
               ]
            },
            "StringEqualsIfExists":{
               "dynamodb:Select":"SPECIFIC_ATTRIBUTES"
            }
         }
      }
   ]
}
```

In addition to granting permissions for specific DynamoDB actions (`Action` element) on the `GameScores` table (`Resource` element), the `Condition` element uses the following condition keys specific to DynamoDB that limit the permissions as follows:

- `dynamodb:LeadingKeys` – This condition key allows users to access only the items where the partition key value matches their user ID. This ID, `${www.amazon.com:user_id}`, is a substitution variable. For more information about substitution variables, see Using web identity federation.
- `dynamodb:Attributes` – This condition key limits access to the specified attributes so that only the actions listed in the permissions policy can return values for these attributes. In

addition, the `StringEqualsIfExists` clause ensures that the app must always provide a list of specific attributes to act upon and that the app can't request all attributes.

When an IAM policy is evaluated, the result is always either true (access is allowed) or false (access is denied). If any part of the `Condition` element is false, the entire policy evaluates to false and access is denied.

> **Important**
>
> If you use `dynamodb:Attributes`, you must specify the names of all of the primary key and index key attributes for the table and any secondary indexes that are listed in the policy. Otherwise, DynamoDB can't use these key attributes to perform the requested action.

IAM policy documents can contain only the following Unicode characters: horizontal tab (U+0009), linefeed (U+000A), carriage return (U+000D), and characters in the range U+0020 to U+00FF.

Example 2: Grant permissions that limit access to items with a specific partition key value

The following permissions policy grants permissions that allow a set of DynamoDB actions on the `GamesScore` table. It uses the `dynamodb:LeadingKeys` condition key to limit user actions only on the items whose `UserID` partition key value matches the Login with Amazon unique user ID for this app.

> **Important**
>
> The list of actions does not include permissions for `Scan` because `Scan` returns all items regardless of the leading keys.

JSON

```
{
   "Version":"2012-10-17",
   "Statement":[
      {
         "Sid":"FullAccessToUserItems",
```

```
         "Effect":"Allow",
         "Action":[
            "dynamodb:GetItem",
            "dynamodb:BatchGetItem",
            "dynamodb:Query",
            "dynamodb:PutItem",
            "dynamodb:UpdateItem",
            "dynamodb:DeleteItem",
            "dynamodb:BatchWriteItem"
         ],
         "Resource":[
            "arn:aws:dynamodb:us-west-2:123456789012:table/GameScores"
         ],
         "Condition":{
            "ForAllValues:StringEquals":{
               "dynamodb:LeadingKeys":[
                  "${www.amazon.com:user_id}"
               ]
            }
         }
      }
   ]
}
```

Note

When using policy variables, you must explicitly specify version 2012-10-17 in the policy. The default version of the access policy language, 2008-10-17, does not support policy variables.

To implement read-only access, you can remove any actions that can modify the data. In the following policy, only those actions that provide read-only access are included in the condition.

JSON

```
{
   "Version":"2012-10-17",
   "Statement":[
      {
```

```
         "Sid":"ReadOnlyAccessToUserItems",
         "Effect":"Allow",
         "Action":[
            "dynamodb:GetItem",
            "dynamodb:BatchGetItem",
            "dynamodb:Query"
         ],
         "Resource":[
            "arn:aws:dynamodb:us-west-2:123456789012:table/GameScores"
         ],
         "Condition":{
            "ForAllValues:StringEquals":{
               "dynamodb:LeadingKeys":[
                  "${www.amazon.com:user_id}"
               ]
            }
         }
      }
   ]
}
```

Important

If you use dynamodb:Attributes, you must specify the names of all of the primary key and index key attributes, for the table and any secondary indexes that are listed in the policy. Otherwise, DynamoDB can't use these key attributes to perform the requested action.

Example 3: Grant permissions that limit access to specific attributes in a table

The following permissions policy allows access to only two specific attributes in a table by adding the dynamodb:Attributes condition key. These attributes can be read, written, or evaluated in a conditional write or scan filter.

JSON

```
{
   "Version":"2012-10-17",
   "Statement":[
```

```
      {
         "Sid":"LimitAccessToSpecificAttributes",
         "Effect":"Allow",
         "Action":[
            "dynamodb:UpdateItem",
            "dynamodb:GetItem",
            "dynamodb:Query",
            "dynamodb:BatchGetItem",
            "dynamodb:Scan"
         ],
         "Resource":[
            "arn:aws:dynamodb:us-west-2:123456789012:table/GameScores"
         ],
         "Condition":{
            "ForAllValues:StringEquals":{
               "dynamodb:Attributes":[
                  "UserId",
                  "TopScore"
               ]
            },
            "StringEqualsIfExists":{
               "dynamodb:Select":"SPECIFIC_ATTRIBUTES",
               "dynamodb:ReturnValues":[
                  "NONE",
                  "UPDATED_OLD",
                  "UPDATED_NEW"
               ]
            }
         }
      }
   ]
}
```

Note

The policy takes an allow list approach, which allows access to a named set of attributes. You can write an equivalent policy that denies access to other attributes instead. We don't recommend this deny list approach. Users can determine the names of these denied attributes by follow the principle of least privilege, as explained in Wikipedia at http://

en.wikipedia.org/wiki/Principle_of_least_privilege, and use an allow list approach to enumerate all of the allowed values, rather than specifying the denied attributes.

This policy doesn't permit `PutItem`, `DeleteItem`, or `BatchWriteItem`. These actions always replace the entire previous item, which would allow users to delete the previous values for attributes that they are not allowed to access.

The `StringEqualsIfExists` clause in the permissions policy ensures the following:

- If the user specifies the `Select` parameter, then its value must be `SPECIFIC_ATTRIBUTES`. This requirement prevents the API action from returning any attributes that aren't allowed, such as from an index projection.
- If the user specifies the `ReturnValues` parameter, then its value must be `NONE`, `UPDATED_OLD`, or `UPDATED_NEW`. This is required because the `UpdateItem` action also performs implicit read operations to check whether an item exists before replacing it, and so that previous attribute values can be returned if requested. Restricting `ReturnValues` in this way ensures that users can only read or write the allowed attributes.
- The `StringEqualsIfExists` clause assures that only one of these parameters — `Select` or `ReturnValues` — can be used per request, in the context of the allowed actions.

The following are some variations on this policy:

- To allow only read actions, you can remove `UpdateItem` from the list of allowed actions. Because none of the remaining actions accept `ReturnValues`, you can remove `ReturnValues` from the condition. You can also change `StringEqualsIfExists` to `StringEquals` because the `Select` parameter always has a value (`ALL_ATTRIBUTES`, unless otherwise specified).
- To allow only write actions, you can remove everything except `UpdateItem` from the list of allowed actions. Because `UpdateItem` does not use the `Select` parameter, you can remove `Select` from the condition. You must also change `StringEqualsIfExists` to `StringEquals` because the `ReturnValues` parameter always has a value (`NONE` unless otherwise specified).
- To allow all attributes whose name matches a pattern, use `StringLike` instead of `StringEquals`, and use a multi-character pattern match wildcard character (*).

Example 4: Grant permissions to prevent updates on certain attributes

The following permissions policy limits user access to updating only the specific attributes identified by the dynamodb:Attributes condition key. The StringNotLike condition prevents an application from updating the attributes specified using the dynamodb:Attributes condition key.

JSON

```
{
   "Version":"2012-10-17",
   "Statement":[
      {
         "Sid":"PreventUpdatesOnCertainAttributes",
         "Effect":"Allow",
         "Action":[
            "dynamodb:UpdateItem"
         ],
         "Resource":"arn:aws:dynamodb:us-west-2:123456789012:table/GameScores",
         "Condition":{
            "ForAllValues:StringNotLike":{
               "dynamodb:Attributes":[
                  "FreeGamesAvailable",
                  "BossLevelUnlocked"
               ]
            },
            "StringEqualsIfExists":{
               "dynamodb:Select":"SPECIFIC_ATTRIBUTES",
               "dynamodb:ReturnValues":[
                  "NONE",
                  "UPDATED_OLD",
                  "UPDATED_NEW"
               ]
            }
         }
      }
   ]
}
```

Note the following:

- The `UpdateItem` action, like other write actions, requires read access to the items so that it can return values before and after the update. In the policy, you limit the action to accessing only the attributes that are allowed to be updated by specifying the `dynamodb:ReturnValues` condition key. The condition key restricts `ReturnValues` in the request to specify only `NONE`, `UPDATED_OLD`, or `UPDATED_NEW` and doesn't include `ALL_OLD` or `ALL_NEW`.
- The `StringEqualsIfExists` operator ensures that if `dynamodb:Select` or `dynamodb:ReturnValues` is present in the request, it must match the specified values. This prevents operations from returning complete items.
- When restricting attribute updates, you should also control what data can be returned to prevent information disclosure of protected attributes.
- The `PutItem` and `DeleteItem` actions replace an entire item, and thus allows applications to modify any attributes. So when limiting an application to updating only specific attributes, you should not grant permission for these APIs.

Example 5: Grant permissions to query only projected attributes in an index

The following permissions policy allows queries on a secondary index (`TopScoreDateTimeIndex`) by using the `dynamodb:Attributes` condition key. The policy also limits queries to requesting only specific attributes that have been projected into the index.

To require the application to specify a list of attributes in the query, the policy also specifies the `dynamodb:Select` condition key to require that the `Select` parameter of the DynamoDB `Query` action is `SPECIFIC_ATTRIBUTES`. The list of attributes is limited to a specific list that is provided using the `dynamodb:Attributes` condition key.

JSON

```
{
   "Version":"2012-10-17",
   "Statement":[
      {
         "Sid":"QueryOnlyProjectedIndexAttributes",
         "Effect":"Allow",
         "Action":[
            "dynamodb:Query"
         ],
         "Resource":[
```

```
            "arn:aws:dynamodb:us-west-2:123456789012:table/GameScores/index/
TopScoreDateTimeIndex"
         ],
         "Condition":{
            "ForAllValues:StringEquals":{
               "dynamodb:Attributes":[
                  "TopScoreDateTime",
                  "GameTitle",
                  "Wins",
                  "Losses",
                  "Attempts"
               ]
            },
            "StringEquals":{
               "dynamodb:Select":"SPECIFIC_ATTRIBUTES"
            }
         }
      }
   ]
}
```

The following permissions policy is similar, but the query must request all of the attributes that have been projected into the index.

JSON

```
{
   "Version":"2012-10-17",
   "Statement":[
      {
         "Sid":"QueryAllIndexAttributes",
         "Effect":"Allow",
         "Action":[
            "dynamodb:Query"
         ],
         "Resource":[
            "arn:aws:dynamodb:us-west-2:123456789012:table/GameScores/index/
TopScoreDateTimeIndex"
         ],
         "Condition":{
            "StringEquals":{
```

```
                "dynamodb:Select":"ALL_PROJECTED_ATTRIBUTES"
            }
         }
      }
   ]
}
```

Example 6: Grant permissions to limit access to certain attributes and partition key values

The following permissions policy allows specific DynamoDB actions (specified in the `Action` element) on a table and a table index (specified in the `Resource` element). The policy uses the `dynamodb:LeadingKeys` condition key to restrict permissions to only the items whose partition key value matches the user's Facebook ID.

JSON

```
{
   "Version":"2012-10-17",
   "Statement":[
      {
         "Sid":"LimitAccessToCertainAttributesAndKeyValues",
         "Effect":"Allow",
         "Action":[
            "dynamodb:UpdateItem",
            "dynamodb:GetItem",
            "dynamodb:Query",
            "dynamodb:BatchGetItem"
         ],
         "Resource":[
            "arn:aws:dynamodb:us-west-2:123456789012:table/GameScores",
            "arn:aws:dynamodb:us-west-2:123456789012:table/GameScores/index/
TopScoreDateTimeIndex"
         ],
         "Condition":{
            "ForAllValues:StringEquals":{
               "dynamodb:LeadingKeys":[
                  "${graph.facebook.com:id}"
               ],
               "dynamodb:Attributes":[
                  "attribute-A",
                  "attribute-B"
```

```
            ]
         },
         "StringEqualsIfExists":{
            "dynamodb:Select":"SPECIFIC_ATTRIBUTES",
            "dynamodb:ReturnValues":[
               "NONE",
               "UPDATED_OLD",
               "UPDATED_NEW"
            ]
         }
      }
   ]
}
```

Note the following:

- Write actions allowed by the policy (`UpdateItem`) can only modify attribute-A or attribute-B.
- Because the policy allows `UpdateItem`, an application can insert new items, and the hidden attributes will be null in the new items. If these attributes are projected into `TopScoreDateTimeIndex`, the policy has the added benefit of preventing queries that cause fetches from the table.
- Applications cannot read any attributes other than those listed in `dynamodb:Attributes`. With this policy in place, an application must set the `Select` parameter to `SPECIFIC_ATTRIBUTES` in read requests, and only attributes in the allow list can be requested. For write requests, the application cannot set `ReturnValues` to `ALL_OLD` or `ALL_NEW` and it cannot perform conditional write operations based on any other attributes.

Example 7: Deny permissions to limit access to specific attributes in a table

The following policy denies access to sensitive attributes and ensures this restriction cannot be bypassed by omitting a projection expression. It allows general access to the `CustomerData` table while explicitly denying access to `SSN` and `CreditCardNumber` attributes.

JSON

```
{
   "Version":"2012-10-17",
```

```
   "Statement":[
      {
         "Effect":"Allow",
         "Action":[
            "dynamodb:GetItem",
            "dynamodb:Query",
            "dynamodb:Scan"
         ],
         "Resource":"arn:aws:dynamodb:us-west-2:123456789012:table/CustomerData"
      },
      {
         "Effect":"Deny",
         "Action":[
            "dynamodb:GetItem",
            "dynamodb:Query",
            "dynamodb:Scan"
         ],
         "Resource":"arn:aws:dynamodb:us-west-2:123456789012:table/CustomerData",
         "Condition":{
            "ForAnyValue:StringEquals":{
               "dynamodb:Attributes":[
                  "SSN",
                  "CreditCardNumber"
               ]
            }
         }
      },
      {
         "Effect":"Deny",
         "Action":[
            "dynamodb:GetItem",
            "dynamodb:Query",
            "dynamodb:Scan"
         ],
         "Resource":"arn:aws:dynamodb:us-west-2:123456789012:table/CustomerData",
         "Condition":{
            "StringNotEqualsIfExists":{
               "dynamodb:Select":"SPECIFIC_ATTRIBUTES"
            }
         }
      }
   ]
}
```

Related topics

- Identity and Access Management for Amazon DynamoDB
- DynamoDB API permissions: Actions, resources, and conditions reference

Using web identity federation

If you are writing an application targeted at large numbers of users, you can optionally use *web identity federation* for authentication and authorization. Web identity federation removes the need for creating individual users. Instead, users can sign in to an identity provider and then obtain temporary security credentials from AWS Security Token Service (AWS STS). The app can then use these credentials to access AWS services.

Web identity federation supports the following identity providers:

- Login with Amazon
- Facebook
- Google

Additional resources for web identity federation

The following resources can help you learn more about web identity federation:

- The post Web Identity Federation using the AWS SDK for .NET on the AWS Developer blog walks through how to use web identity federation with Facebook. It includes code snippets in C# that show how to assume an IAM role with web identity and how to use temporary security credentials to access an AWS resource.
- The AWS Mobile SDK for iOS and the AWS Mobile SDK for Android contain sample apps. They include code that shows how to invoke the identity providers, and then how to use the information from these providers to get and use temporary security credentials.
- The article Web Identity Federation with Mobile Applications discusses web identity federation and shows an example of how to use web identity federation to access an AWS resource.

Example policy for web identity federation

To show how you can use web identity federation with DynamoDB, revisit the *GameScores* table that was introduced in Using IAM policy conditions for fine-grained access control. Here is the primary key for *GameScores*.

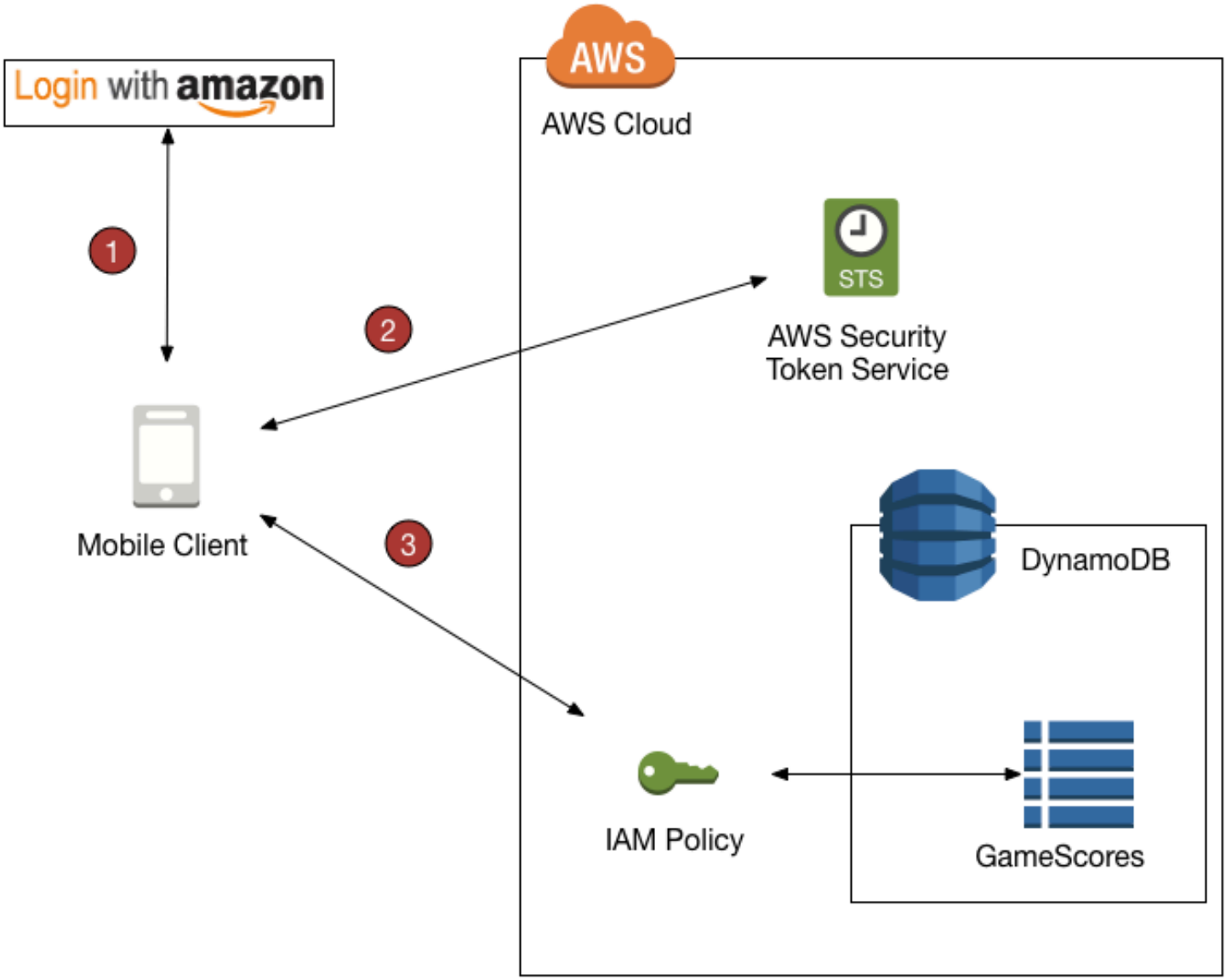
Table Name	Primary Key Type	Partition Key Name and Type	Sort Key Name and Type
GameScores (UserId, GameTitle, ...)	Composite	Attribute Name: UserId Type: String	Attribute Name: GameTitle Type: String

Now suppose that a mobile gaming app uses this table, and that app needs to support thousands, or even millions, of users. At this scale, it becomes very difficult to manage individual app users, and to guarantee that each user can only access their own data in the *GameScores* table. Fortunately, many users already have accounts with a third-party identity provider, such as Facebook, Google, or Login with Amazon. So it makes sense to use one of these providers for authentication tasks.

To do this using web identity federation, the app developer must register the app with an identity provider (such as Login with Amazon) and obtain a unique app ID. Next, the developer needs to create an IAM role. (For this example, this role is named *GameRole*.) The role must have an IAM policy document attached to it, specifying the conditions under which the app can access *GameScores* table.

When a user wants to play a game, they sign in to their Login with Amazon account from within the gaming app. The app then calls AWS Security Token Service (AWS STS), providing the Login with Amazon app ID and requesting membership in *GameRole*. AWS STS returns temporary AWS credentials to the app and allows it to access the *GameScores* table, subject to the *GameRole* policy document.

The following diagram shows how these pieces fit together.



Web identity federation overview

1. The app calls a third-party identity provider to authenticate the user and the app. The identity provider returns a web identity token to the app.
2. The app calls AWS STS and passes the web identity token as input. AWS STS authorizes the app and gives it temporary AWS access credentials. The app is allowed to assume an IAM role (*GameRole*) and access AWS resources in accordance with the role's security policy.
3. The app calls DynamoDB to access the *GameScores* table. Because it has assumed the *GameRole*, the app is subject to the security policy associated with that role. The policy document prevents the app from accessing data that does not belong to the user.

Once again, here is the security policy for *GameRole* that was shown in Using IAM policy conditions for fine-grained access control:

JSON

```
{
   "Version":"2012-10-17",
   "Statement":[
      {
         "Sid":"AllowAccessToOnlyItemsMatchingUserID",
         "Effect":"Allow",
         "Action":[
            "dynamodb:GetItem",
            "dynamodb:BatchGetItem",
            "dynamodb:Query",
            "dynamodb:PutItem",
            "dynamodb:UpdateItem",
            "dynamodb:DeleteItem",
            "dynamodb:BatchWriteItem"
         ],
         "Resource":[
            "arn:aws:dynamodb:us-west-2:123456789012:table/GameScores"
         ],
         "Condition":{
            "ForAllValues:StringEquals":{
               "dynamodb:LeadingKeys":[
                  "${www.amazon.com:user_id}"
               ],
               "dynamodb:Attributes":[
                  "UserId",
                  "GameTitle",
                  "Wins",
                  "Losses",
                  "TopScore",
                  "TopScoreDateTime"
               ]
            },
            "StringEqualsIfExists":{
               "dynamodb:Select":"SPECIFIC_ATTRIBUTES"
            }
         }
      }
   ]
```

```
}
```

The `Condition` clause determines which items in *GameScores* are visible to the app. It does this by comparing the Login with Amazon ID to the `UserId` partition key values in `GameScores`. Only the items belonging to the current user can be processed using one of DynamoDB actions that are listed in this policy. Other items in the table cannot be accessed. Furthermore, only the specific attributes listed in the policy can be accessed.

Preparing to use web identity federation

If you are an application developer and want to use web identity federation for your app, follow these steps:

1. **Sign up as a developer with a third-party identity provider.** The following external links provide information about signing up with supported identity providers:
   - Login with Amazon Developer Center
   - Registration on the Facebook site
   - Using OAuth 2.0 to Access Google APIs on the Google site
2. **Register your app with the identity provider.** When you do this, the provider gives you an ID that's unique to your app. If you want your app to work with multiple identity providers, you need to obtain an app ID from each provider.
3. **Create one or more IAM roles.** You need one role for each identity provider for each app. For example, you might create a role that can be assumed by an app where the user signed in using Login with Amazon, a second role for the same app where the user has signed in using Facebook, and a third role for the app where users sign in using Google.

   As part of the role creation process, you need to attach an IAM policy to the role. Your policy document should define the DynamoDB resources required by your app, and the permissions for accessing those resources.

For more information, see About Web Identity Federation in *IAM User Guide*.

> **Note**
>
> As an alternative to AWS Security Token Service, you can use Amazon Cognito. Amazon Cognito is the preferred service for managing temporary credentials for mobile apps. For more information, see Getting credentials in the *Amazon Cognito Developer Guide.*

Generating an IAM policy using the DynamoDB console

The DynamoDB console can help you create an IAM policy for use with web identity federation. To do this, you choose a DynamoDB table and specify the identity provider, actions, and attributes to be included in the policy. The DynamoDB console then generates a policy that you can attach to an IAM role.

1. Sign in to the AWS Management Console and open the DynamoDB console at https://console.aws.amazon.com/dynamodb/.
2. In the navigation pane, choose **Tables**.
3. In the list of tables, choose the table for which you want to create the IAM policy.
4. Select the **Actions** button, and choose **Create Access Control Policy**.
5. Choose the identity provider, actions, and attributes for the policy.

   When the settings are as you want them, choose **Generate Policy**. The generated policy appears.
6. Choose **See Documentation**, and follow the steps required to attach the generated policy to an IAM role.

Writing your app to use web identity federation

To use web identity federation, your app must assume the IAM role that you created. From that point on, the app honors the access policy that you attached to the role.

At runtime, if your app uses web identity federation, it must follow these steps:

1. **Authenticate with a third-party identity provider.** Your app must call the identity provider using an interface that they provide. The exact way in which you authenticate the user depends on the provider and on what platform your app is running. Typically, if the user is not already signed in, the identity provider takes care of displaying a sign-in page for that provider.

After the identity provider authenticates the user, the provider returns a web identity token to your app. The format of this token depends on the provider, but is typically a very long string of characters.

2. **Obtain temporary AWS security credentials.** To do this, your app sends a `AssumeRoleWithWebIdentity` request to AWS Security Token Service (AWS STS). This request contains the following:
   - The web identity token from the previous step
   - The app ID from the identity provider
   - The Amazon Resource Name (ARN) of the IAM role that you created for this identity provider for this app

   AWS STS returns a set of AWS security credentials that expire after a certain amount of time (3,600 seconds, by default).

   The following is a sample request and response from a `AssumeRoleWithWebIdentity` action in AWS STS. The web identity token was obtained from the Login with Amazon identity provider.

```
GET / HTTP/1.1
Host: sts.amazonaws.com
Content-Type: application/json; charset=utf-8
URL: https://sts.amazonaws.com/?ProviderId=www.amazon.com
&DurationSeconds=900&Action=AssumeRoleWithWebIdentity
&Version=2011-06-15&RoleSessionName=web-identity-federation
&RoleArn=arn:aws:iam::123456789012:role/GameRole
&WebIdentityToken=Atza|IQEBLjAsAhQluyKqyBiYZ8-kclvGTYM81e...(remaining characters
 omitted)
```

```
<AssumeRoleWithWebIdentityResponse
  xmlns="https://sts.amazonaws.com/doc/2011-06-15/">
  <AssumeRoleWithWebIdentityResult>
    <SubjectFromWebIdentityToken>amzn1.account.AGJZDKHJKAUUSW6C44CHPEXAMPLE</
SubjectFromWebIdentityToken>
    <Credentials>
      <SessionToken>AQoDYXdzEMf//////////wEa8AP6nNDwcSLnf+cHupC...(remaining
 characters omitted)</SessionToken>
      <SecretAccessKey>8Jhi60+EWUUbbUShTEsjTxqQtM8UKvsM6XAjdA==</SecretAccessKey>
      <Expiration>2013-10-01T22:14:35Z</Expiration>
      <AccessKeyId>06198791C436IEXAMPLE</AccessKeyId>
```

```
    </Credentials>
    <AssumedRoleUser>
      <Arn>arn:aws:sts::123456789012:assumed-role/GameRole/web-identity-federation</
Arn>
      <AssumedRoleId>AROAJU4SA2VW5SZRF2YMG:web-identity-federation</AssumedRoleId>
    </AssumedRoleUser>
  </AssumeRoleWithWebIdentityResult>
  <ResponseMetadata>
    <RequestId>c265ac8e-2ae4-11e3-8775-6969323a932d</RequestId>
  </ResponseMetadata>
</AssumeRoleWithWebIdentityResponse>
```

3. **Access AWS resources.** The response from AWS STS contains information that your app requires in order to access DynamoDB resources:

   - The `AccessKeyID`, `SecretAccessKey`, and `SessionToken` fields contain security credentials that are valid for this user and this app only.
   - The `Expiration` field signifies the time limit for these credentials, after which they are no longer valid.
   - The `AssumedRoleId` field contains the name of a session-specific IAM role that has been assumed by the app. The app honors the access controls in the IAM policy document for the duration of this session.
   - The `SubjectFromWebIdentityToken` field contains the unique ID that appears in an IAM policy variable for this particular identity provider. The following are the IAM policy variables for supported providers, and some example values for them:

Policy Variable	Example Value
`${www.amazon.com:user_id}`	`amzn1.account.AGJZDKHJKAUUSW6C44CHPEXAMPLE`
`${graph.facebook.com:id}`	`123456789`
`${accounts.google.com:sub}`	`12345678901234567890 1`

For example IAM policies where these policy variables are used, see Example policies: Using conditions for fine-grained access control.

For more information about how AWS STS generates temporary access credentials, see Requesting Temporary Security Credentials in *IAM User Guide*.

DynamoDB API permissions: Actions, resources, and conditions reference

When you are setting up Identity and Access Management for Amazon DynamoDB and writing a permissions policy that you can attach to an IAM identity (identity-based policies), you can use the list of Actions, resources, and condition keys for Amazon DynamoDB in the *IAM User Guide* as a reference. The page lists each DynamoDB API operation, the corresponding actions for which you can grant permissions to perform the action, and the AWS resource for which you can grant the permissions. You specify the actions in the policy's `Action` field, and you specify the resource value in the policy's `Resource` field.

You can use AWS-wide condition keys in your DynamoDB policies to express conditions. For a complete list of AWS-wide keys, see the IAM JSON policy elements reference in the *IAM User Guide*.

In addition to the AWS-wide condition keys, DynamoDB has its own specific keys that you can use in conditions. For more information, see Using IAM policy conditions for fine-grained access control.

Related topics

- Identity and Access Management for Amazon DynamoDB
- Using IAM policy conditions for fine-grained access control

Compliance validation by industry for DynamoDB

To learn whether an AWS service is within the scope of specific compliance programs, see AWS services in Scope by Compliance Program and choose the compliance program that you are interested in. For general information, see AWS Compliance Programs.

You can download third-party audit reports using AWS Artifact. For more information, see Downloading Reports in AWS Artifact.

Your compliance responsibility when using AWS services is determined by the sensitivity of your data, your company's compliance objectives, and applicable laws and regulations. For more information about your compliance responsibility when using AWS services, see AWS Security Documentation.

Resilience and disaster recovery in Amazon DynamoDB

The AWS global infrastructure is built around AWS Regions and Availability Zones. AWS Regions provide multiple physically separated and isolated Availability Zones, which are connected with low-latency, high-throughput, and highly redundant networking. With Availability Zones, you can design and operate applications and databases that automatically fail over between Availability Zones without interruption. Availability Zones are more highly available, fault tolerant, and scalable than traditional single or multiple data center infrastructures.

If you need to replicate your data or applications over greater geographic distances, use AWS Local Regions. An AWS Local Region is a single data center designed to complement an existing AWS Region. Like all AWS Regions, AWS Local Regions are completely isolated from other AWS Regions.

For more information about AWS Regions and Availability Zones, see AWS global infrastructure.

Amazon DynamoDB automatically replicates your data across three Availability Zones in a Region, providing built-in high durability and a 99.99% availability SLA. In addition, DynamoDB offers several features to help support your data resiliency and backup needs.

On-demand backup and restore

DynamoDB provides on-demand backup capability. It allows you to create full backups of your tables for long-term retention and archival. For more information, see On-Demand backup and restore for DynamoDB.

Point-in-time recovery

Point-in-time recovery helps protect your DynamoDB tables from accidental write or delete operations. With point in time recovery, you don't have to worry about creating, maintaining, or scheduling on-demand backups. For more information, see Point-in-time recovery for DynamoDB.

Global tables that sync across AWS regions

DynamoDB automatically spreads the data and traffic for your tables over a sufficient number of servers to handle your throughput and storage requirements, while maintaining consistent and fast performance. All of your data is stored on solid-state disks (SSDs) and is automatically replicated across multiple Availability Zones in an AWS Region, providing built-in high availability and data durability. You can use global tables to keep DynamoDB tables in sync across AWS Regions.

Infrastructure security in Amazon DynamoDB

As a managed service, Amazon DynamoDB is protected by the AWS global network security procedures that are described in Infrastructure protection located in the AWS Well-Architected Framework.

You use AWS published API calls to access DynamoDB through the network. Clients can use TLS (Transport Layer Security) version 1.2 or 1.3. Clients must also support cipher suites with perfect forward secrecy (PFS) such as Ephemeral Diffie-Hellman (DHE) or Elliptic Curve Diffie-Hellman Ephemeral (ECDHE). Most modern systems such as Java 7 and later support these modes. Additionally, requests must be signed using an access key ID and a secret access key that is associated with an IAM principal. Or you can use the AWS Security Token Service (AWS STS) to generate temporary security credentials to sign requests.

You can also use a virtual private cloud (VPC) endpoint for DynamoDB to enable Amazon EC2 instances in your VPC to use their private IP addresses to access DynamoDB with no exposure to the public internet. For more information, see Using Amazon VPC endpoints to access DynamoDB.

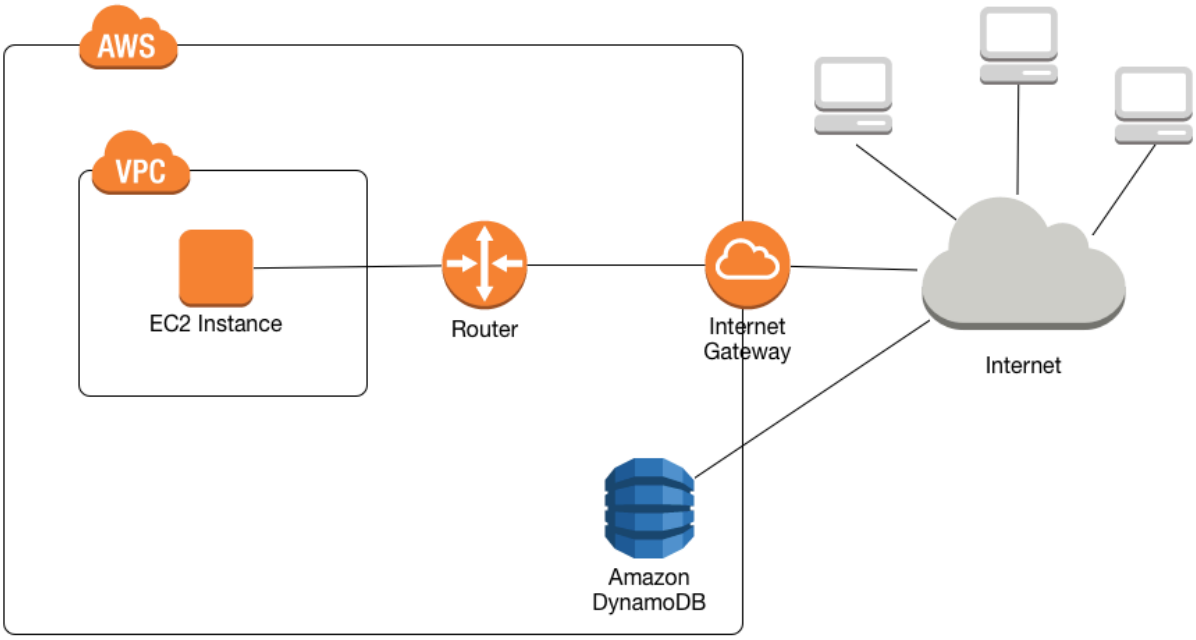
Using Amazon VPC endpoints to access DynamoDB

For security reasons, many AWS customers run their applications within an Amazon Virtual Private Cloud environment (Amazon VPC). With Amazon VPC, you can launch Amazon EC2 instances into a virtual private cloud, which is logically isolated from other networks—including the public internet. With an Amazon VPC, you have control over its IP address range, subnets, routing tables, network gateways, and security settings.

> **Note**
>
> If you created your AWS account after December 4, 2013, then you already have a default VPC in each AWS Region. A default VPC is ready for you to use—you can immediately start using it without having to perform any additional configuration steps.
> For more information, see Default VPC and Default Subnets in the *Amazon VPC User Guide*.

To access the public internet, your VPC must have an internet gateway—a virtual router that connects your VPC to the internet. This allows applications running on Amazon EC2 in your VPC to access internet resources, such as Amazon DynamoDB.

By default, communications to and from DynamoDB use the HTTPS protocol, which protects network traffic by using SSL/TLS encryption. The following diagram shows an Amazon EC2 instance in a VPC accessing DynamoDB, by having DynamoDB use an internet gateway rather than VPC endpoints.



Many customers have legitimate privacy and security concerns about sending and receiving data across the public internet. You can address these concerns by using a virtual private network (VPN) to route all DynamoDB network traffic through your own corporate network infrastructure. However, this approach can introduce bandwidth and availability challenges.

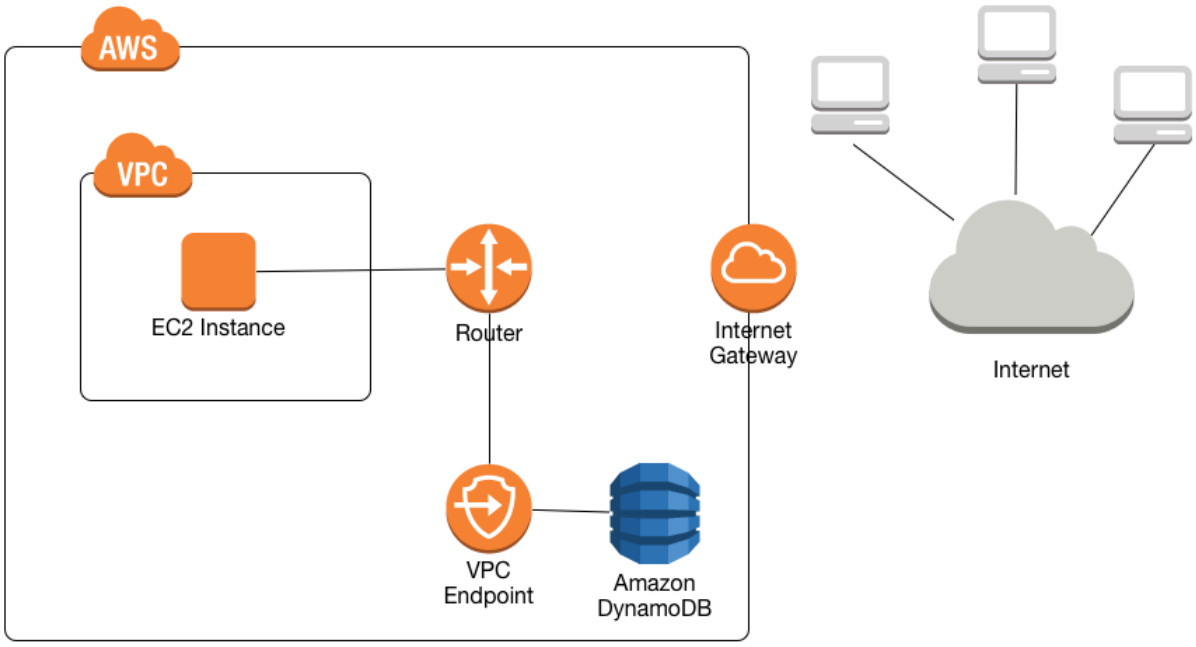
VPC endpoints for DynamoDB can alleviate these challenges. A *VPC endpoint* for DynamoDB enables Amazon EC2 instances in your VPC to use their private IP addresses to access DynamoDB with no exposure to the public internet. Your EC2 instances do not require public IP addresses, and you don't need an internet gateway, a NAT device, or a virtual private gateway in your VPC. You use endpoint policies to control access to DynamoDB. Traffic between your VPC and the AWS service does not leave the Amazon network.

> **Note**
>
> Even when you use public IP addresses, all VPC communication between instances and services hosted in AWS is kept private within the AWS network. Packets that originate from

the AWS network with a destination on the AWS network stay on the AWS global network, except traffic to or from AWS China Regions.

When you create a VPC endpoint for DynamoDB, any requests to a DynamoDB endpoint within the Region (for example, *dynamodb.us-west-2.amazonaws.com*) are routed to a private DynamoDB endpoint within the Amazon network. You don't need to modify your applications running on EC2 instances in your VPC. The endpoint name remains the same, but the route to DynamoDB stays entirely within the Amazon network, and does not access the public internet.

The following diagram shows how an EC2 instance in a VPC can use a VPC endpoint to access DynamoDB.



For more information, see the section called “Tutorial: Using a VPC endpoint for DynamoDB”.

Sharing Amazon VPC endpoints and DynamoDB

In order to enable access to the DynamoDB service through a VPC subnet's gateway endpoint, you must have owner account permissions for that VPC subnet.

Once the VPC subnet’s gateway endpoint has been granted access to DynamoDB, any AWS account with access to that subnet can use DynamoDB. This means all account users within the VPC

subnet can use any DynamoDB tables which they have access to. This includes DynamoDB tables associated with a different account than the VPC subnet. The VPC subnet owner can still restrict any particular user within the subnet from using the DynamoDB service through the gateway endpoint, at their discretion.

Tutorial: Using a VPC endpoint for DynamoDB

This section walks you through setting up and using a VPC endpoint for DynamoDB.

Topics

- Step 1: Launch an Amazon EC2 instance
- Step 2: Configure your Amazon EC2 instance
- Step 3: Create a VPC endpoint for DynamoDB
- Step 4: (Optional) Clean up

Step 1: Launch an Amazon EC2 instance

In this step, you launch an Amazon EC2 instance in your default Amazon VPC. You can then create and use a VPC endpoint for DynamoDB.

1. Open the Amazon EC2 console at https://console.aws.amazon.com/ec2/.
2. Choose **Launch Instance** and do the following:

   Step 1: Choose an Amazon Machine Image (AMI)

   - At the top of the list of AMIs, go to **Amazon Linux AMI** and choose **Select**.

   Step 2: Choose an Instance Type

   - At the top of the list of instance types, choose **t2.micro**.
   - Choose **Next: Configure Instance Details**.

   Step 3: Configure Instance Details

   - Go to **Network** and choose your default VPC.

     Choose **Next: Add Storage**.

Step 4: Add Storage

- Skip this step by choosing **Next: Tag Instance**.

Step 5: Tag Instance

- Skip this step by choosing **Next: Configure Security Group**.

Step 6: Configure Security Group

- Choose **Select an existing security group**.
- In the list of security groups, choose **default**. This is the default security group for your VPC.
- Choose **Next: Review and Launch**.

Step 7: Review Instance Launch

- Choose **Launch**.

3. In the **Select an existing key pair or create a new key pair** window, do one of the following:
   - If you do not have an Amazon EC2 key pair, choose **Create a new key pair** and follow the instructions. You will be asked to download a private key file (*.pem* file); you will need this file later when you log in to your Amazon EC2 instance.
   - If you already have an existing Amazon EC2 key pair, go to **Select a key pair** and choose your key pair from the list. You must already have the private key file (*.pem* file) available in order to log in to your Amazon EC2 instance.
4. When you have configured your key pair, choose **Launch Instances**.
5. Return to the Amazon EC2 console home page and choose the instance that you launched. In the lower pane, on the **Description** tab, find the **Public DNS** for your instance. For example: `ec2-00-00-00-00.us-east-1.compute.amazonaws.com`.

   Make a note of this public DNS name, because you will need it in the next step in this tutorial (Step 2: Configure your Amazon EC2 instance).

> **Note**
>
> It will take a few minutes for your Amazon EC2 instance to become available. Before you go on to the next step, ensure that the **Instance State** is `running` and that all of its **Status Checks** have passed.

Step 2: Configure your Amazon EC2 instance

When your Amazon EC2 instance is available, you will be able to log into it and prepare it for first use.

> **Note**
>
> The following steps assume that you are connecting to your Amazon EC2 instance from a computer running Linux. For other ways to connect, see Connect to Your Linux Instance in the Amazon EC2 User Guide.

1. You will need to authorize inbound SSH traffic to your Amazon EC2 instance. To do this, you will create a new EC2 security group, and then assign the security group to your EC2 instance.

   a. In the navigation pane, choose **Security Groups**.

   b. Choose **Create Security Group**. In the **Create Security Group** window, do the following:

   - **Security group name**—type a name for your security group. For example: `my-ssh-access`
   - **Description**—type a short description for the security group.
   - **VPC**—choose your default VPC.
   - In the **Security group rules** section, choose **Add Rule** and do the following:
     - **Type**—choose SSH.
     - **Source**—choose My IP.

   When the settings are as you want them, choose **Create**.

   c. In the navigation pane, choose **Instances**.

d. Choose the Amazon EC2 instance that you launched in Step 1: Launch an Amazon EC2 instance.

e. Choose **Actions** --> **Networking** --> **Change Security Groups**.

f. In the **Change Security Groups**, select the security group that you created earlier in this procedure (for example: `my-ssh-access`). The existing `default` security group should also be selected. When the settings are as you want them, choose **Assign Security Groups**.

2. Use the `ssh` command to log in to your Amazon EC2 instance, as in the following example.

```
ssh -i my-keypair.pem ec2-user@public-dns-name
```

You will need to specify your private key file (*.pem* file) and the public DNS name of your instance. (See Step 1: Launch an Amazon EC2 instance).

The login ID is `ec2-user`. No password is required.

3. Configure your AWS credentials as shown in the following example. Enter your AWS access key ID, secret key, and default Region name when prompted.

```
aws configure
```

```
AWS Access Key ID [None]: AKIAIOSFODNN7EXAMPLE
AWS Secret Access Key [None]: wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
Default region name [None]: us-east-1
Default output format [None]:
```

You are now ready to create a VPC endpoint for DynamoDB.

Step 3: Create a VPC endpoint for DynamoDB

In this step, you will create a VPC endpoint for DynamoDB and test it to make sure that it works.

1. Before you begin, verify that you can communicate with DynamoDB using its public endpoint.

```
aws dynamodb list-tables
```

The output will show a list of DynamoDB tables that you currently own. (If you don't have any tables, the list will be empty.).

2. Verify that DynamoDB is an available service for creating VPC endpoints in the current AWS Region. (The command is shown in bold text, followed by example output.)

```
aws ec2 describe-vpc-endpoint-services
```

```
{
    "ServiceNames": [
        "com.amazonaws.us-east-1.s3",
        "com.amazonaws.us-east-1.dynamodb"
    ]
}
```

   In the example output, DynamoDB is one of the services available, so you can proceed with creating a VPC endpoint for it.

3. Determine your VPC identifier.

```
aws ec2 describe-vpcs
```

```
{
    "Vpcs": [
        {
            "VpcId": "vpc-0bbc736e",
            "InstanceTenancy": "default",
            "State": "available",
            "DhcpOptionsId": "dopt-8454b7e1",
            "CidrBlock": "172.31.0.0/16",
            "IsDefault": true
        }
    ]
}
```

   In the example output, the VPC ID is `vpc-0bbc736e`.

4. Create the VPC endpoint. For the `--vpc-id` parameter, specify the VPC ID from the previous step. Use the `--route-table-ids` parameter to associate the endpoint with your route tables.

```
aws ec2 create-vpc-endpoint --vpc-id vpc-0bbc736e --service-name com.amazonaws.us-
east-1.dynamodb --route-table-ids rtb-11aa22bb
```

```
{
    "VpcEndpoint": {
        "PolicyDocument": "{\"Version\":\"2008-10-17\",\"Statement\":[{\"Effect\":
\"Allow\",\"Principal\":\"*\",\"Action\":\"*\",\"Resource\":\"*\"}]}",
        "VpcId": "vpc-0bbc736e",
        "State": "available",
        "ServiceName": "com.amazonaws.us-east-1.dynamodb",
        "RouteTableIds": [
            "rtb-11aa22bb"
        ],
        "VpcEndpointId": "vpce-9b15e2f2",
        "CreationTimestamp": "2017-07-26T22:00:14Z"
    }
}
```

5. Verify that you can access DynamoDB through the VPC endpoint.

```
aws dynamodb list-tables
```

If you want, you can try some other AWS CLI commands for DynamoDB. For more information, see the AWS CLI Command Reference.

Step 4: (Optional) Clean up

If you want to delete the resources you have created in this tutorial, follow these procedures:

To remove your VPC endpoint for DynamoDB

1. Log in to your Amazon EC2 instance.
2. Determine the VPC endpoint ID.

```
aws ec2 describe-vpc-endpoints
```

```
{
    "VpcEndpoint": {
        "PolicyDocument": "{\"Version\":\"2008-10-17\",\"Statement\":[{\"Effect\":
\"Allow\",\"Principal\":\"*\",\"Action\":\"*\",\"Resource\":\"*\"}]}",
        "VpcId": "vpc-0bbc736e",
        "State": "available",
        "ServiceName": "com.amazonaws.us-east-1.dynamodb",
```

```
        "RouteTableIds": [],
        "VpcEndpointId": "vpce-9b15e2f2",
        "CreationTimestamp": "2017-07-26T22:00:14Z"
    }
}
```

In the example output, the VPC endpoint ID is `vpce-9b15e2f2`.

3. Delete the VPC endpoint.

```
aws ec2 delete-vpc-endpoints --vpc-endpoint-ids vpce-9b15e2f2
```

```
{
    "Unsuccessful": []
}
```

The empty array `[]` indicates success (there were no unsuccessful requests).

To terminate your Amazon EC2 instance

1. Open the Amazon EC2 console at https://console.aws.amazon.com/ec2/.
2. In the navigation pane, choose **Instances**.
3. Choose your Amazon EC2 instance.
4. Choose **Actions**, **Instance State**, **Terminate**.
5. In the confirmation window, choose **Yes, Terminate**.

AWS PrivateLink for DynamoDB

With AWS PrivateLink for DynamoDB, you can provision *interface Amazon VPC endpoints* (interface endpoints) in your virtual private cloud (Amazon VPC). These endpoints are directly accessible from applications that are on premises over VPN and Direct Connect, or in a different AWS Region over Amazon VPC peering. Using AWS PrivateLink and interface endpoints, you can simplify private network connectivity from your applications to DynamoDB.

Applications in your VPC do not need public IP addresses to communicate with DynamoDB using VPC interface endpoints for DynamoDB operations. Interface endpoints are represented by one or more elastic network interfaces (ENIs) that are assigned private IP addresses from subnets in your

Amazon VPC. Requests to DynamoDB over interface endpoints stay on the Amazon network. You can also access interface endpoints in your Amazon VPC from on-premises applications through AWS Direct Connect or AWS Virtual Private Network (Site-to-Site VPN). For more information about how to connect your Amazon VPC with your on-premises network, see the Direct Connect User Guide and the AWS Site-to-Site VPN User Guide.

For general information about interface endpoints, see Interface Amazon VPC endpoints (AWS PrivateLink) in the *AWS PrivateLink Guide*. AWS PrivateLink is also supported for Amazon DynamoDB Streams endpoints. For more information, see the section called "AWS PrivateLink for DynamoDB Streams".

Topics

- Types of Amazon VPC endpoints for Amazon DynamoDB
- Considerations when using AWS PrivateLink for Amazon DynamoDB
- Creating an Amazon VPC endpoint
- Accessing Amazon DynamoDB interface endpoints
- Accessing DynamoDB tables and control API operations from DynamoDB interface endpoints
- Updating an on-premises DNS configuration
- Creating an Amazon VPC endpoint policy for DynamoDB
- Using DynamoDB endpoints with AWS Management Console Private Access
- AWS PrivateLink for DynamoDB Streams
- Using AWS PrivateLink for DynamoDB Accelerator (DAX)

Types of Amazon VPC endpoints for Amazon DynamoDB

You can use two types of Amazon VPC endpoints to access Amazon DynamoDB: *gateway endpoints* and *interface endpoints* (by using AWS PrivateLink). A *gateway endpoint* is a gateway that you specify in your route table to access DynamoDB from your Amazon VPC over the AWS network. *Interface endpoints* extend the functionality of gateway endpoints by using private IP addresses to route requests to DynamoDB from within your Amazon VPC, on premises, or from an Amazon VPC in another AWS Region by using Amazon VPC peering or AWS Transit Gateway. For more information, see What is Amazon VPC peering? and Transit Gateway vs Amazon VPC peering.

Interface endpoints are compatible with gateway endpoints. If you have an existing gateway endpoint in the Amazon VPC, you can use both types of endpoints in the same Amazon VPC.

Gateway endpoints for DynamoDB	Interface endpoints for DynamoDB
In both cases, your network traffic remains on the AWS network.	
Use Amazon DynamoDB public IP addresses	Use private IP addresses from your Amazon VPC to access Amazon DynamoDB
Do not allow access from on premises	Allow access from on premises
Do not allow access from another AWS Region	Allow access from an Amazon VPC endpoint in another AWS Region by using Amazon VPC peering or AWS Transit Gateway
Not billed	Billed

For more information about gateway endpoints, see Gateway Amazon VPC endpoints in the *AWS PrivateLink Guide*.

Considerations when using AWS PrivateLink for Amazon DynamoDB

Amazon VPC considerations apply to AWS PrivateLink for Amazon DynamoDB. For more information, see Interface endpoint considerations and AWS PrivateLink quotas in the *AWS PrivateLink Guide*. In addition, the following restrictions apply.

AWS PrivateLink for Amazon DynamoDB does not support the following:

- Transport Layer Security (TLS) 1.1
- Private and Hybrid Domain Name System (DNS) services

Important

Do not create private hosted zones to override DynamoDB endpoint DNS names (such as `dynamodb.region.amazonaws.com` or `*.region.amazonaws.com`) to route traffic to your interface endpoints. DynamoDB DNS configurations may change over time. Custom DNS overrides are not compatible with these changes and can cause requests to unexpectedly route over public IP addresses instead of your interface endpoints.

To access DynamoDB through AWS PrivateLink, configure your clients to use the Amazon VPC endpoint URL directly (for example, `https://vpce-1a2b3c4d-5e6f.dynamodb.region.vpce.amazonaws.com`).

You can submit up to 50,000 requests per second for each AWS PrivateLink endpoint that you enable.

Note

Network connectivity timeouts to AWS PrivateLink endpoints are not within the scope of DynamoDB error responses and need to be appropriately handled by your applications connecting to the PrivateLink endpoints.

Creating an Amazon VPC endpoint

To create an Amazon VPC interface endpoint, see Create an Amazon VPC endpoint in the *AWS PrivateLink Guide*.

Accessing Amazon DynamoDB interface endpoints

When you create an interface endpoint, DynamoDB generates two types of endpoint-specific, DynamoDB DNS names: *Regional* and *Zonal*.

- A *Regional* DNS name includes a unique Amazon VPC endpoint ID, a service identifier, the AWS Region, and `vpce.amazonaws.com` in its name. For example, for Amazon VPC endpoint ID *vpce-1a2b3c4d*, the DNS name generated might be similar to *vpce-1a2b3c4d-5e6f*`.dynamodb.us-east-1.vpce.amazonaws.com`.
- A *Zonal* DNS name includes the Availability Zone—for example, *vpce-1a2b3c4d-5e6f*`-us-east-1a.dynamodb.us-east-1.vpce.amazonaws.com`. You might use this option if your architecture isolates Availability Zones. For example, you could use it for fault containment or to reduce Regional data transfer costs.

> **Note**
>
> To achieve optimal reliability, we recommend deploying your service across a minimum of three availability zones.

Accessing DynamoDB tables and control API operations from DynamoDB interface endpoints

You can use the AWS CLI or AWS SDKs to access DynamoDB tables and control API operations through DynamoDB interface endpoints.

AWS CLI examples

To access DynamoDB tables or DynamoDB control API operations through DynamoDB interface endpoints in AWS CLI commands, use the `--region` and `--endpoint-url` parameters.

Example: Create a VPC endpoint

```
aws ec2 create-vpc-endpoint \
--region us-east-1 \
--service-name com.amazonaws.us-east-1.dynamodb \
--vpc-id client-vpc-id \
--subnet-ids client-subnet-id \
--vpc-endpoint-type Interface \
--security-group-ids client-sg-id
```

Example: Modify a VPC endpoint

```
aws ec2 modify-vpc-endpoint \
--region us-east-1 \
--vpc-endpoint-id client-vpc-endpoint-id \
--policy-document policy-document \ #example optional parameter
--add-security-group-ids security-group-ids \ #example optional parameter
# any additional parameters needed, see Privatelink documentation for more details
```

Example: List tables using an endpoint URL

In the following example, replace the Region `us-east-1` and the DNS name of the VPC endpoint ID `vpce-1a2b3c4d-5e6f.dynamodb.us-east-1.vpce.amazonaws.com` with your own information.

```
aws dynamodb --region us-east-1 --endpoint https://vpce-1a2b3c4d-5e6f.dynamodb.us-
east-1.vpce.amazonaws.com list-tables
```

AWS SDK examples

To access DynamoDB tables or DynamoDB control API operations through DynamoDB interface endpoints when using the AWS SDKs, update your SDKs to the latest version. Then, configure your clients to use an endpoint URL for accessing a table or DynamoDB control API operation through DynamoDB interface endpoints.

SDK for Python (Boto3)

Example: Use an endpoint URL to access a DynamoDB table

In the following example, replace the Region `us-east-1` and VPC endpoint ID `https://vpce-1a2b3c4d-5e6f.dynamodb.us-east-1.vpce.amazonaws.com` with your own information.

```
ddb_client = session.client(
service_name='dynamodb',
region_name='us-east-1',
endpoint_url='https://vpce-1a2b3c4d-5e6f.dynamodb.us-east-1.vpce.amazonaws.com'
)
```

SDK for Java 1.x

Example: Use an endpoint URL to access a DynamoDB table

In the following example, replace the Region `us-east-1` and VPC endpoint ID `https://vpce-1a2b3c4d-5e6f.dynamodb.us-east-1.vpce.amazonaws.com` with your own information.

```
//client build with endpoint config
final AmazonDynamoDB dynamodb =
 AmazonDynamoDBClientBuilder.standard().withEndpointConfiguration(
```

```
        new AwsClientBuilder.EndpointConfiguration(
                "https://vpce-1a2b3c4d-5e6f.dynamodb.us-east-1.vpce.amazonaws.com",
                Regions.DEFAULT_REGION.getName()
        )
).build();
```

SDK for Java 2.x

Example: Use an endpoint URL to access DynamoDB table

In the following example, replace the Region us-east-1 and VPC endpoint ID https://vpce-1a2b3c4d-5e6f.dynamodb.us-east-1.vpce.amazonaws.com with your own information.

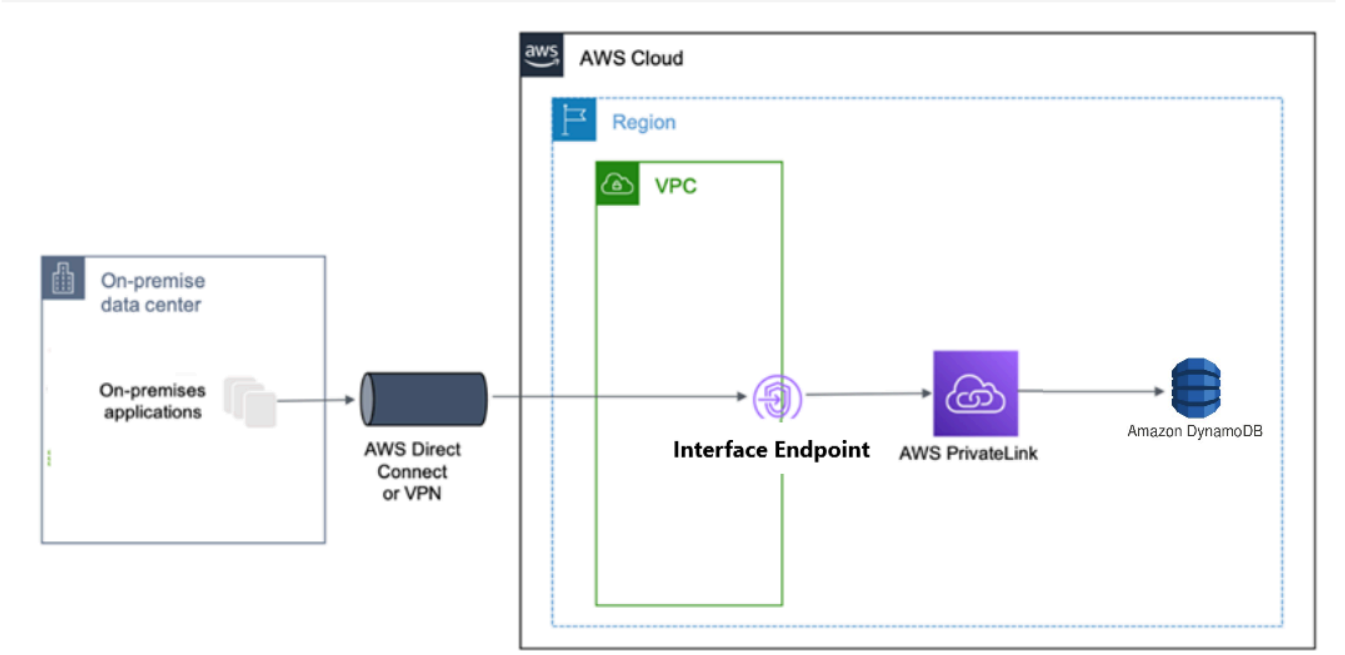
```
Region region = Region.US_EAST_1;
dynamoDbClient = DynamoDbClient.builder().region(region)
.endpointOverride(URI.create("https://vpce-1a2b3c4d-5e6f.dynamodb.us-
east-1.vpce.amazonaws.com"))
.build()
```

Updating an on-premises DNS configuration

When using endpoint-specific DNS names to access the interface endpoints for DynamoDB, you don't have to update your on-premises DNS resolver. You can resolve the endpoint-specific DNS name with the private IP address of the interface endpoint from the public DynamoDB DNS domain.

Using interface endpoints to access DynamoDB without a gateway endpoint or an internet gateway in the Amazon VPC

Interface endpoints in your Amazon VPC can route both in-Amazon VPC applications and on-premises applications to DynamoDB over the Amazon network, as illustrated in the following diagram.

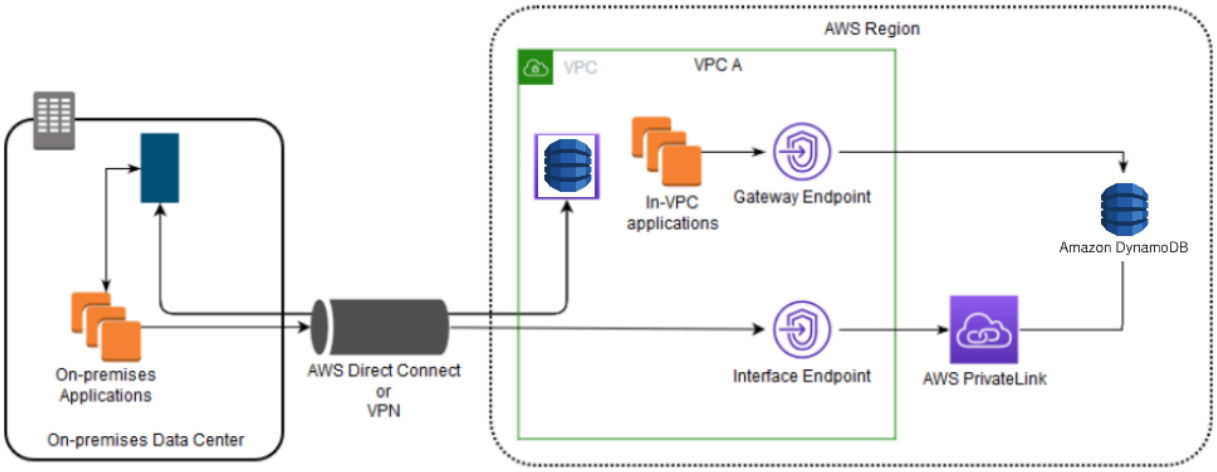


The diagram illustrates the following:

- Your on-premises network uses Direct Connect or Site-to-Site VPN to connect to Amazon VPC A.
- Your applications on-premises and in Amazon VPC A use endpoint-specific DNS names to access DynamoDB through the DynamoDB interface endpoint.
- On-premises applications send data to the interface endpoint in the Amazon VPC through Direct Connect (or Site-to-Site VPN). AWS PrivateLink moves the data from the interface endpoint to DynamoDB over the AWS network.
- In-Amazon VPC applications also send traffic to the interface endpoint. AWS PrivateLink moves the data from the interface endpoint to DynamoDB over the AWS network.

Using gateway endpoints and interface endpoints together in the same Amazon VPC to access DynamoDB

You can create interface endpoints and retain the existing gateway endpoint in the same Amazon VPC, as the following diagram shows. By taking this approach, you allow in-Amazon VPC applications to continue accessing DynamoDB through the gateway endpoint, which is not billed. Then, only your on-premises applications would use interface endpoints to access DynamoDB. To access DynamoDB this way, you must update your on-premises applications to use endpoint-specific DNS names for DynamoDB.



The diagram illustrates the following:

- On-premises applications use endpoint-specific DNS names to send data to the interface endpoint within the Amazon VPC through Direct Connect (or Site-to-Site VPN). AWS PrivateLink moves the data from the interface endpoint to DynamoDB over the AWS network.
- Using default Regional DynamoDB names, in-Amazon VPC applications send data to the gateway endpoint that connects to DynamoDB over the AWS network.

For more information about gateway endpoints, see Gateway Amazon VPC endpoints in the *Amazon VPC User Guide*.

Creating an Amazon VPC endpoint policy for DynamoDB

You can attach an endpoint policy to your Amazon VPC endpoint that controls access to DynamoDB. The policy specifies the following information:

- The AWS Identity and Access Management (IAM) principal that can perform actions
- The actions that can be performed
- The resources on which actions can be performed

Topics

- Example: Restricting access to a specific table from an Amazon VPC endpoint

Example: Restricting access to a specific table from an Amazon VPC endpoint

You can create an endpoint policy that restricts access to only specific DynamoDB tables. This type of policy is useful if you have other AWS services in your Amazon VPC that use tables. The following table policy restricts access to only the *DOC-EXAMPLE-TABLE*. To use this endpoint policy, replace *DOC-EXAMPLE-TABLE* with the name of your table.

JSON

```
{
"Version":"2012-10-17",
  "Id": "Policy1216114807515",
  "Statement": [
    { "Sid": "Access-to-specific-table-only",
      "Principal": "*",
      "Action": [
        "dynamodb:GetItem",
        "dynamodb:PutItem"
      ],
      "Effect": "Allow",
      "Resource": ["arn:aws:dynamodb:us-east-1:111122223333:table/DOC-EXAMPLE-
TABLE",
                   "arn:aws:dynamodb:us-east-1:111122223333:table/DOC-EXAMPLE-
TABLE/*"]
    }
  ]
}
```

Using DynamoDB endpoints with AWS Management Console Private Access

You must set up DNS configuration for DynamoDB and DynamoDB Streams when using VPC endpoints with the DynamoDB console in AWS Management Console Private Access.

To configure DynamoDB to be accessible in AWS Management Console Private Access, you must create the following two VPC endpoints:

- `com.amazonaws.<region>.dynamodb`
- `com.amazonaws.<region>.dynamodb-streams`

When you create the VPC endpoints, navitage to the Route53 console and create a private hosted zone for DynamoDB using the regional endpoint `dynamodb.us-east-1.amazonaws.com`.

Create the following two alias records in the private hosted zone:

- `dynamodb.<region>.amazonaws.com` that routes traffic to the VPC endpoint `com.amazonaws.<region>.dynamodb`.
- `streams.dynamodb.<region>.amazonaws.com` that routes traffic to the VPC endpoint `com.amazonaws.<region>.dynamodb-streams`.

AWS PrivateLink for DynamoDB Streams

With AWS PrivateLink for Amazon DynamoDB Streams, you can provision interface Amazon VPC endpoints (interface endpoints) in your virtual private cloud (Amazon VPC). These endpoints are directly accessible from applications that are on premises over VPN and Direct Connect, or in a different AWS Region over Amazon VPC peering. Using AWS PrivateLink and interface endpoints, you can simplify private network connectivity from your applications to DynamoDB Streams.

Applications in your Amazon VPC do not need public IP addresses to communicate with DynamoDB Streams using Amazon VPC interface endpoints for DynamoDB Streams operations. Interface endpoints are represented by one or more elastic network interfaces (ENIs) that are assigned private IP addresses from subnets in your Amazon VPC. Requests to DynamoDB Streams over interface endpoints stay on the Amazon network. You can also access interface endpoints in your Amazon VPC from on-premises applications through Direct Connect or AWS Virtual Private Network (AWS VPN). For more information about how to connect your AWS Virtual Private Network with your on-premises network, see the *Direct Connect User Guide* and the *AWS Site-to-Site VPN User Guide*.

For general information about interface endpoints, see Interface Amazon VPC endpoints (AWS PrivateLink).

> **Note**
>
> Only interface endpoints are supported for DynamoDB Streams. Gateway endpoints are not supported.

Topics

- Considerations when using AWS PrivateLink for Amazon DynamoDB Streams
- Creating an Amazon VPC endpoint
- Accessing Amazon DynamoDB Streams interface endpoints
- Accessing DynamoDB Streams API operations from DynamoDB Streams interface endpoints
- AWS SDK examples
- Creating an Amazon VPC endpoint policy for DynamoDB Streams
- Using DynamoDB endpoints with AWS Management Console Private Access

Considerations when using AWS PrivateLink for Amazon DynamoDB Streams

Amazon VPC considerations apply to AWS PrivateLink for Amazon DynamoDB Streams. For more information, see interface endpoint considerations and AWS PrivateLink quotas. The following restrictions apply.

AWS PrivateLink for Amazon DynamoDB Streams doesn't support the following:

- Transport Layer Security (TLS) 1.1
- Private and Hybrid Domain Name System (DNS) services

Important

Do not create private hosted zones to override DynamoDB Streams endpoint DNS names to route traffic to your interface endpoints. DynamoDB DNS configurations may change over time and custom DNS overrides can cause requests to unexpectedly route over public IP addresses instead of your interface endpoints.
To access DynamoDB Streams through AWS PrivateLink, configure your clients to use the Amazon VPC endpoint URL directly (for example, `https://vpce-1a2b3c4d-5e6f.streams.dynamodb.region.vpce.amazonaws.com`).

Note

Network connectivity timeouts to AWS PrivateLink endpoints are not within the scope of DynamoDB Streams error responses and need to be appropriately handled by your applications connecting to the AWS PrivateLink endpoints.

Creating an Amazon VPC endpoint

To create an Amazon VPC interface endpoint, see Create an Amazon VPC endpoint in the *AWS PrivateLink Guide*.

Accessing Amazon DynamoDB Streams interface endpoints

When you create an interface endpoint, DynamoDB generates two types of endpoint-specific, DynamoDB Streams DNS names: *Regional* and *Zonal*.

- A *Regional* DNS name includes a unique Amazon VPC endpoint ID, a service identifier, the AWS Region, and `vpce.amazonaws.com` in its name. For example, for Amazon VPC endpoint ID *`vpce-1a2b3c4d`*, the DNS name generated might be similar to *`vpce-1a2b3c4d-5e6f`*`.streams.dynamodb.us-east-1.vpce.amazonaws.com`.
- A *Zonal* DNS name includes the Availability Zone—for example, *`vpce-1a2b3c4d-5e6f`***`-us-east-1a.`**`streams.dynamodb.us-east-1.vpce.amazonaws.com`. You might use this option if your architecture isolates Availability Zones. For example, you could use it for fault containment or to reduce Regional data transfer costs.

Accessing DynamoDB Streams API operations from DynamoDB Streams interface endpoints

You can use the AWS CLI or AWS SDKs to access DynamoDB Streams API operations through DynamoDB Streams interface endpoints.

AWS CLI examples

To access DynamoDB Streams or API operations through DynamoDB Streams interface endpoints in AWS CLI commands, use the `--region` and `--endpoint-url` parameters.

Example: Create a VPC endpoint

```
aws ec2 create-vpc-endpoint \
--region us-east-1 \
--service-name com.amazonaws.us-east-1.dynamodb-streams \
--vpc-id client-vpc-id \
--subnet-ids client-subnet-id \
--vpc-endpoint-type Interface \
--security-group-ids client-sg-id
```

Example: Modify a VPC endpoint

```
aws ec2 modify-vpc-endpoint \
--region us-east-1 \
--vpc-endpoint-id client-vpc-endpoint-id \
--policy-document policy-document \ #example optional parameter
--add-security-group-ids security-group-ids \ #example optional parameter
# any additional parameters needed, see Privatelink documentation for more details
```

Example: List streams using an endpoint URL

In the following example, replace the Region `us-east-1` and the DNS name of the VPC endpoint ID `vpce-1a2b3c4d-5e6f.streams.dynamodb.us-east-1.vpce.amazonaws.com` with your own information.

```
aws dynamodbstreams --region us-east-1 –endpoint https://
vpce-1a2b3c4d-5e6f.streams.dynamodb.us-east-1.vpce.amazonaws.com list-streams
```

AWS SDK examples

To access Amazon DynamoDB Streams API operations through DynamoDB Streams interface endpoints when using the AWS SDKs, update your SDKs to the latest version. Then, configure your clients to use an endpoint URL for DynamoDB Streams API operation through DynamoDB Streams interface endpoints.

SDK for Python (Boto3)

Example: Use an endpoint URL to access a DynamoDB stream

In the following example, replace the Region `us-east-1` and VPC endpoint ID `https://vpce-1a2b3c4d-5e6f.streams.dynamodb.us-east-1.vpce.amazonaws.com` with your own information.

```
ddb_streams_client = session.client(
service_name='dynamodbstreams',
region_name='us-east-1',
endpoint_url='https://vpce-1a2b3c4d-5e6f.streams.dynamodb.us-
east-1.vpce.amazonaws.com'
)
```

SDK for Java 1.x

Example: Use an endpoint URL to access a DynamoDB stream

In the following example, replace the Region `us-east-1` and VPC endpoint ID `https://vpce-1a2b3c4d-5e6f.streams.dynamodb.us-east-1.vpce.amazonaws.com` with your own information.

```
//client build with endpoint config
final AmazonDynamoDBStreams dynamodbstreams =
 AmazonDynamoDBStreamsClientBuilder.standard().withEndpointConfiguration(
        new AwsClientBuilder.EndpointConfiguration(
                "https://vpce-1a2b3c4d-5e6f.streams.dynamodb.us-
east-1.vpce.amazonaws.com",
                Regions.DEFAULT_REGION.getName()
        )
).build();
```

SDK for Java 2.x

Example: Use an endpoint URL to access DynamoDB stream

In the following example, replace the Region `us-east-1` and VPC endpoint ID `https://vpce-1a2b3c4d-5e6f.streams.dynamodb.us-east-1.vpce.amazonaws.com` with your own information.

```
Region region = Region.US_EAST_1;
dynamoDbStreamsClient = DynamoDbStreamsClient.builder().region(region)
.endpointOverride(URI.create("https://vpce-1a2b3c4d-5e6f.streams.dynamodb.us-
east-1.vpce.amazonaws.com"))
.build()
```

Creating an Amazon VPC endpoint policy for DynamoDB Streams

You can attach an endpoint policy to your Amazon VPC endpoint that controls access to DynamoDB Streams. The policy specifies the following information:

- The AWS Identity and Access Management (IAM) principal that can perform actions
- The actions that can be performed
- The resources on which actions can be performed

Topics

- Example: Restricting access to a specific stream from an Amazon VPC endpoint

Example: Restricting access to a specific stream from an Amazon VPC endpoint

You can create an endpoint policy that restricts access to only specific DynamoDB Streams. This type of policy is useful if you have other AWS services in your Amazon VPC that use DynamoDB Streams. The following stream policy restricts access to only the stream *2025-02-20T11:22:33.444* attached to *DOC-EXAMPLE-TABLE*. To use this endpoint policy, replace *DOC-EXAMPLE-TABLE* with the name of your table and *2025-02-20T11:22:33.444* with the stream label.

JSON

```
{
"Version":"2012-10-17",
  "Id": "Policy1216114807515",
  "Statement": [
    { "Sid": "Access-to-specific-stream-only",
      "Principal": "*",
      "Action": [
        "dynamodb:DescribeStream",
        "dynamodb:GetRecords"
      ],
      "Effect": "Allow",
      "Resource": ["arn:aws:dynamodb:us-east-1:111122223333:table/table-name/
stream/2025-02-20T11:22:33.444"]
    }
  ]
}
```

> **Note**
>
> Gateway endpoints aren't supported in DynamoDB Streams.

Using DynamoDB endpoints with AWS Management Console Private Access

You must set up DNS configuration for DynamoDB and DynamoDB Streams when using VPC endpoints with the DynamoDB console in AWS Management Console Private Access.

To configure DynamoDB to be accessible in AWS Management Console Private Access, you must create the following two VPC endpoints:

- `com.amazonaws.<region>.dynamodb`
- `com.amazonaws.<region>.dynamodb-streams`

When you create the VPC endpoints, navitage to the Route53 console and create a private hosted zone for DynamoDB using the regional endpoint `dynamodb.us-east-1.amazonaws.com`.

Create the following two alias records in the private hosted zone:

- `dynamodb.<region>.amazonaws.com` that routes traffic to the VPC endpoint `com.amazonaws.<region>.dynamodb`.
- `streams.dynamodb.<region>.amazonaws.com` that routes traffic to the VPC endpoint `com.amazonaws.<region>.dynamodb-streams`.

Using AWS PrivateLink for DynamoDB Accelerator (DAX)

AWS PrivateLink for DynamoDB Accelerator (DAX) enables you to securely access DAX management APIs such as `CreateCluster`, `DescribeClusters`, and `DeleteCluster` over private IP addresses within your virtual private cloud (VPC). This feature enables you to access DAX services privately from your applications without exposing traffic to the public internet.

DAX PrivateLink supports dual-stack endpoints (`dax.{region}.api.aws`), enabling both IPv4 and IPv6 connectivity. With AWS PrivateLink for DAX, customers can access the service using private DNS names. The dual-stack endpoint support ensures transparent connectivity while maintaining network privacy. This allows you to access DAX through both public internet and VPC endpoints without making any changes to your SDK configuration.

Considerations when using AWS PrivateLink for DynamoDB Accelerator (DAX)

When implementing AWS PrivateLink for DynamoDB Accelerator (DAX), several important considerations must be taken into account.

Before you set up an interface endpoint for DAX consider the following:

- DAX interface endpoints only support access to the DAX management APIs within the same AWS Region. You can't use an interface endpoint to access DAX management APIs in other Regions.
- To access the AWS Management Console privately for DAX management, you may need to create additional VPC endpoints for services like `com.amazonaws.region.console` and related services.
- You are charged for creating and using an interface endpoint to DAX. For pricing information, see AWS PrivateLink pricing.

How AWS PrivateLink works with DAX

When you create an interface endpoint for DAX:

1. AWS creates an endpoint network interface in each subnet you enable for the interface endpoint.
2. These are requester-managed network interfaces that serve as entry points for traffic destined for DAX.
3. You can then access DAX through private IP addresses within your VPC.
4. This architecture allows you to use VPC security groups to manage access to the endpoints.
5. Applications can access both DynamoDB and DAX through their respective interface endpoints within a VPC, while also allowing on-premises applications to connect via Direct Connect or VPN.
6. This provides a consistent connectivity model across both services, simplifies architecture, and improves security by keeping traffic within the AWS network.

Creating Interface Endpoints for DAX

You can create an interface endpoint to connect to DAX using the AWS Management Console, AWS SDK, CloudFormation, or the AWS API.

To create an interface endpoint for DAX using the console

1. Navigate to the Amazon VPC console at https://console.aws.amazon.com/vpc/.
2. In the navigation pane, choose **Endpoints**.
3. Choose **Create Endpoint**.

4. For **Service category**, choose **AWS services** and for **Service Name**, search for and select `com.amazonaws.region.dax`.
5. For **VPC**, select the VPC from which you want to access DAX and for **Subnets**, select the subnets where AWS will create the endpoint network interfaces.
6. For **Security groups**, select or create security groups to associate with the endpoint network interfaces.
7. For **Policy**, keep the default **Full Access** or customize as needed.
8. Select **Enable DNS Name** to enable private DNS for the endpoint. Keep the private DNS name enabled to prevent changes in the SDK configuration. When enabled, your applications can continue using the standard service DNS name (example: `dax.region.amazonaws.com`). AWS creates a private hosted zone in your VPC that resolves this name to your endpoint's private IP address.

   > **Note**
   >
   > Use Regional DNS names if required. Using zonal DNS names isn't recommended. Also, select subnets from 3 or more AZs to ensure maximum availability through PrivateLink.

9. Choose **Create endpoint**.

To create an interface endpoint for DAX using the AWS CLI

Use the `create-vpc-endpoint` command with the `vpc-endpoint-type` parameter set to `Interface` and the `service-name` parameter set to `com.amazonaws.region.dax`.

```
aws ec2 create-vpc-endpoint \
    --vpc-id vpc-ec43eb89 \
    --vpc-endpoint-type Interface \
    --service-name com.amazonaws.us-east-1.dax \
    --subnet-ids subnet-abcd1234 subnet-1a2b3c4d \
    --security-group-ids sg-1a2b3c4d \
    --private-dns-enabled
```

Additional resources

For more information about AWS PrivateLink and VPC endpoints, see the following resources:

- AWS PrivateLink for DynamoDB

- AWS PrivateLink for DynamoDB Streams
- Connect your VPC to services using AWS PrivateLink
- Simplify private connectivity to DynamoDB with AWS PrivateLink
- AWS PrivateLink Whitepaper

Configuration and vulnerability analysis in Amazon DynamoDB

AWS handles basic security tasks like guest operating system (OS) and database patching, firewall configuration, and disaster recovery. These procedures have been reviewed and certified by the appropriate third parties. For more details, see the following resources:

- Compliance validation for Amazon DynamoDB
- Shared responsibility model
- Amazon Web Services: Overview of security processes(whitepaper)

The following security best practices also address configuration and vulnerability analysis in Amazon DynamoDB:

- Monitor DynamoDB compliance with AWS Config Rules
- Monitor DynamoDB configuration with AWS Config

Security best practices for Amazon DynamoDB

Amazon DynamoDB provides a number of security features to consider as you develop and implement your own security policies. The following best practices are general guidelines and don't represent a complete security solution. Because these best practices might not be appropriate or sufficient for your environment, treat them as helpful considerations rather than prescriptions.

Topics

- DynamoDB preventative security best practices
- DynamoDB detective security best practices

DynamoDB preventative security best practices

The following best practices can help you anticipate and prevent security incidents in Amazon DynamoDB.

Encryption at rest

DynamoDB encrypts at rest all user data stored in tables, indexes, streams, and backups using encryption keys stored in AWS Key Management Service (AWS KMS). This provides an additional layer of data protection by securing your data from unauthorized access to the underlying storage .

You can specify whether DynamoDB should use an AWS owned key (default encryption type), an AWS managed key, or a customer managed key to encrypt user data. For more information, see Amazon DynamoDB Encryption at Rest.

Use IAM roles to authenticate access to DynamoDB

For users, applications, and other AWS services to access DynamoDB, they must include valid AWS credentials in their AWS API requests. You should not store AWS credentials directly in the application or EC2 instance. These are long-term credentials that are not automatically rotated, and therefore could have significant business impact if they are compromised. An IAM role enables you to obtain temporary access keys that can be used to access AWS services and resources.

For more information, see Identity and Access Management for Amazon DynamoDB.

Use IAM policies for DynamoDB base authorization

When granting permissions, you decide who is getting them, which DynamoDB APIs they are getting permissions for, and the specific actions you want to allow on those resources. Implementing least privilege is key in reducing security risk and the impact that can result from errors or malicious intent.

Attach permissions policies to IAM identities (that is, users, groups, and roles) and thereby grant permissions to perform operations on DynamoDB resources.

You can do this by using the following:

- AWS Managed (predefined) policies
- Customer managed policies

Use IAM policy conditions for fine-grained access control

When you grant permissions in DynamoDB, you can specify conditions that determine how a permissions policy takes effect. Implementing least privilege is key in reducing security risk and the impact that can result from errors or malicious intent.

You can specify conditions when granting permissions using an IAM policy. For example, you can do the following:

- Grant permissions to allow users read-only access to certain items and attributes in a table or a secondary index.
- Grant permissions to allow users write-only access to certain attributes in a table, based upon the identity of that user.

For more information, see Using IAM Policy Conditions for Fine-Grained Access Control.

Use a VPC endpoint and policies to access DynamoDB

If you only require access to DynamoDB from within a virtual private cloud (VPC), you should use a VPC endpoint to limit access from only the required VPC. Doing this prevents that traffic from traversing the open internet and being subject to that environment.

Using a VPC endpoint for DynamoDB allows you to control and limit access using the following:

- VPC endpoint policies – These policies are applied on the DynamoDB VPC endpoint. They allow you to control and limit API access to the DynamoDB table.
- IAM policies – By using the `aws:sourceVpce` condition on policies attached to users, groups, or roles, you can enforce that all access to the DynamoDB table is via the specified VPC endpoint.

For more information, see Endpoints for Amazon DynamoDB.

Consider client-side encryption

We recommend that you plan your encryption strategy before implementing your table in DynamoDB. If you store sensitive or confidential data in DynamoDB, consider including client-side encryption in your plan. This way you can encrypt data as close as possible to its origin, and ensure its protection throughout its lifecycle. Encrypting your sensitive data in transit and at rest helps ensure that your plaintext data isn't available to any third party.

The AWS Database Encryption SDK for DynamoDB is a software library that helps you protect your table data before you send it to DynamoDB. It encrypts, signs, verifies, and decrypts your DynamoDB table items. You control which attributes are encrypted and signed.

Primary Key considerations

Do not use sensitive names or sensitive plaintext data in your Primary Key for your table and Global Secondary Indexes. Key names will show up in your table definition. For example, the Primary Key names are accessible to anyone with permissions to call DescribeTable. Key values can show up in your AWS CloudTrail and other logs. Additionally, DynamoDB uses the key values to distribute data and route requests and AWS administrators may observe the values to maintain the health of the service.

If you need to use sensitive data in your table or GSI key values, we recommend using end-to-end client encryption. This allows you to perform key-value references to your data while ensuring that it never appears unencrypted in your DynamoDB related logs. One way to accomplish this is to use the AWS Database Encryption SDK for DynamoDB, but that is not required. If you use your own solution, we should always use a sufficiently secure encryption algorithm. You should not use a non-cryptographic option like a hash, as they are not considered sufficiently secure in most situations.

If your Primary Key key names are sensitive, we recommend using `pk` and `sk` instead. This is a general best practice which leaves your Partition Key design flexible.

Always consult your security experts or AWS account team if you are concerned about what the right choice would be.

DynamoDB detective security best practices

The following best practices for Amazon DynamoDB can help you detect potential security weaknesses and incidents.

Use AWS CloudTrail to monitor AWS managed KMS key usage

If you are using an AWS managed key for encryption at rest, usage of this key is logged into AWS CloudTrail. CloudTrail provides visibility into user activity by recording actions taken on your account. CloudTrail records important information about each action, including who made the request, the services used, the actions performed, parameters for the actions, and the response elements returned by the AWS service. This information helps you track changes made to your AWS resources and troubleshoot operational issues. CloudTrail makes it easier to ensure compliance with internal policies and regulatory standards.

You can use CloudTrail to audit key usage. CloudTrail creates log files that contain a history of AWS API calls and related events for your account. These log files include all AWS KMS API requests made using the AWS Management Console, AWS SDKs, and command line tools, in addition to those made through integrated AWS services. You can use these log files to get information about when the KMS key was used, the operation that was requested, the identity of the requester, the IP address that the request came from, and so on. For more information, see Logging AWS KMS API Calls with AWS CloudTrail and the AWS CloudTrail User Guide.

Monitor DynamoDB operations using CloudTrail

CloudTrail can monitor both control plane events and data plane events. Control plane operations let you create and manage DynamoDB tables. They also let you work with indexes, streams, and other objects that are dependent on tables. Data plane operations let you perform create, read, update, and delete (also called *CRUD*) actions on data in a table. Some data plane operations also let you read data from a secondary index. To enable logging of data plane events in CloudTrail, you'll need to enable logging of data plane API activity in CloudTrail. See Logging data events for trails for more information.

When activity occurs in DynamoDB, that activity is recorded in a CloudTrail event along with other AWS service events in the event history. For more information, see Logging DynamoDB Operations by Using AWS CloudTrail. You can view, search, and download recent events in your AWS account. For more information, see Viewing Events with CloudTrail Event History in the *AWS CloudTrail User Guide*.

For an ongoing record of events in your AWS account, including events for DynamoDB, create a trail. A trail enables CloudTrail to deliver log files to an Amazon Simple Storage Service (Amazon S3) bucket. By default, when you create a trail on the console, the trail applies to all AWS Regions. The trail logs events from all Regions in the AWS partition and delivers the log files to the S3 bucket that you specify. Additionally, you can configure other AWS services to further analyze and act upon the event data collected in CloudTrail logs.

Use DynamoDB Streams to monitor data plane operations

DynamoDB is integrated with AWS Lambda so that you can create triggers—pieces of code that automatically respond to events in DynamoDB Streams. With triggers, you can build applications that react to data modifications in DynamoDB tables.

If you enable DynamoDB Streams on a table, you can associate the stream Amazon Resource Name (ARN) with a Lambda function that you write. Immediately after an item in the table is modified, a new record appears in the table's stream. AWS Lambda polls the stream and invokes

your Lambda function synchronously when it detects new stream records. The Lambda function can perform any actions that you specify, such as sending a notification or initiating a workflow.

For an example, see Tutorial: Using AWS Lambda with Amazon DynamoDB Streams. This example receives a DynamoDB event input, processes the messages that it contains, and writes some of the incoming event data to Amazon CloudWatch Logs.

Monitor DynamoDB configuration with AWS Config

Using AWS Config, you can continuously monitor and record configuration changes of your AWS resources. You can also use AWS Config to inventory your AWS resources. When a change from a previous state is detected, an Amazon Simple Notification Service (Amazon SNS) notification can be delivered for you to review and take action. Follow the guidance in Setting Up AWS Config with the Console, ensuring that DynamoDB resource types are included.

You can configure AWS Config to stream configuration changes and notifications to an Amazon SNS topic. For example, when a resource is updated, you can get a notification sent to your email, so that you can view the changes. You can also be notified when AWS Config evaluates your custom or managed rules against your resources.

For an example, see Notifications that AWS Config Sends to an Amazon SNS topic in the *AWS Config Developer Guide*.

Monitor DynamoDB compliance with AWS Config rules

AWS Config continuously tracks the configuration changes that occur among your resources. It checks whether these changes violate any of the conditions in your rules. If a resource violates a rule, AWS Config flags the resource and the rule as noncompliant.

By using AWS Config to evaluate your resource configurations, you can assess how well your resource configurations comply with internal practices, industry guidelines, and regulations. AWS Config provides AWS managed rules, which are predefined, customizable rules that AWS Config uses to evaluate whether your AWS resources comply with common best practices.

Tag your DynamoDB resources for identification and automation

You can assign metadata to your AWS resources in the form of tags. Each tag is a simple label consisting of a customer-defined key and an optional value that can make it easier to manage, search for, and filter resources.

Tagging allows for grouped controls to be implemented. Although there are no inherent types of tags, they enable you to categorize resources by purpose, owner, environment, or other criteria. The following are some examples:

- Security – Used to determine requirements such as encryption.
- Confidentiality – An identifier for the specific data-confidentiality level a resource supports.
- Environment – Used to distinguish between development, test, and production infrastructure.

For more information, see AWS Tagging Strategies and Tagging for DynamoDB.

Monitor your usage of Amazon DynamoDB as it relates to security best practices by using AWS Security Hub CSPM.

Security Hub CSPM uses security controls to evaluate resource configurations and security standards to help you comply with various compliance frameworks.

For more information about using Security Hub CSPM to evaluate DynamoDB resources, see Amazon DynamoDB controls in the *AWS Security Hub CSPM User Guide*.

Monitoring and logging in DynamoDB

Monitoring is an important part of maintaining the reliability, availability, and performance of DynamoDB and your AWS solutions. You should collect monitoring data from all parts of your AWS solutions so you can easily debug a multi-point failure.

Topics

- Monitoring plan
- Performance baseline
- Integrated services
- Automated monitoring tools
- Monitoring metrics in DynamoDB with Amazon CloudWatch
- Logging DynamoDB operations by using AWS CloudTrail
- Analyzing data access using CloudWatch contributor insights for DynamoDB

Monitoring plan

Before you start monitoring DynamoDB, create a monitoring plan that includes answers to the following questions:

- What are your monitoring goals?
- What resources will you monitor?
- How often will you monitor these resources?
- What monitoring tools will you use?
- Who will perform the monitoring tasks?
- Who should be notified when something goes wrong?

Performance baseline

Establish a baseline for normal DynamoDB performance in your environment, by measuring performance at various times and under different load conditions. As you monitor DynamoDB, you should consider storing historical monitoring data. This stored data will give you a baseline from which to compare current performance data, identify normal performance patterns and

performance anomalies, and devise methods to address issues. To establish a baseline you should, at a minimum, monitor the following items:

- The number of read or write capacity units consumed over the specified time period, so you can track how much of your provisioned throughput is used.
- Requests that exceeded a table's provisioned write or read capacity during the specified time period, so you can determine which requests exceed the provisioned throughput quotas of a table.
- System errors, so you can determine if any requests resulted in an error.

Integrated services

DynamoDB automatically monitors your tables on your behalf and reports metrics through Amazon CloudWatch. Additionally, DynamoDB integrates with the following AWS services to help you monitor and troubleshoot your DynamoDB resources.

- AWS CloudTrail captures API calls and related events made by or on behalf of your AWS account and delivers the log files to an Amazon S3 bucket that you specify. For more information, see Logging DynamoDB operations by using AWS CloudTrail.
- Contributor Insights is a diagnostic tool for identifying the most frequently accessed and throttled keys in your table or index at a glance. For more information, see Analyzing data access using CloudWatch contributor insights for DynamoDB.

Automated monitoring tools

AWS provides various tools that you can use to monitor DynamoDB. We recommend that you automate monitoring tasks as much as possible. You can use the following automated monitoring tools to watch DynamoDB and report when something is wrong:

- **Amazon CloudWatch alarms** – Watch a single metric over a time period that you specify, and perform one or more actions based on the value of the metric relative to a given threshold over a number of time periods.

  The action is a notification sent to an Amazon Simple Notification Service (Amazon SNS) topic or Amazon EC2 Auto Scaling policy. Amazon CloudWatch alarms do not invoke actions simply because they are in a particular state; the state must have changed and been maintained for a

specified number of periods. For more information, see Monitoring metrics in DynamoDB with Amazon CloudWatch.

- **AWS CloudTrail log monitoring** – Share log files between accounts, monitor AWS CloudTrail log files in real time by sending them to AWS CloudTrail Logs, write log processing applications in Java, and validate that your log files haven't changed after delivery by AWS CloudTrail. For more information, see What is Amazon CloudWatch Logs in the *AWS CloudTrail User Guide*.

Monitoring metrics in DynamoDB with Amazon CloudWatch

You can monitor DynamoDB using CloudWatch, which collects and processes raw data from DynamoDB into readable, near real-time metrics. These statistics are retained for a period of time, so you can access historical information for a better perspective on how your web application or service is performing. By default, DynamoDB metric data is sent to CloudWatch automatically. For more information, see What is Amazon CloudWatch? and Metrics retention in the *Amazon CloudWatch User Guide*.

Topics

- How do I use DynamoDB metrics?
- Viewing metrics in the CloudWatch console
- Viewing metrics in the AWS CLI
- DynamoDB Metrics and dimensions
- Creating CloudWatch alarms in DynamoDB

How do I use DynamoDB metrics?

The metrics reported by DynamoDB provide information that you can analyze in different ways. The following list shows some common uses for the metrics. These are suggestions to get you started, not a comprehensive list.

How do I use DynamoDB metrics?

How can I?	Relevant metrics
How can I monitor the rate of TTL deletions on my table?	You can monitor `TimeToLiveDeletedItemCount` over the specified time period, to track the rate of TTL deletions on your table. For an example of a server-less application using the

How can I?	Relevant metrics
	`TimeToLiveDeletedItemCount` metric, see Automatically archive items to S3 using DynamoDB time to live (TTL) with AWS Lambda and Amazon Data Firehose.
How can I determine how much of my provisioned throughput is being used?	You can monitor `ConsumedReadCapacityUnits` or `ConsumedWriteCapacityUnits` over the specified time period, to track how much of your provisioned throughput is being used.
How can I determine which requests exceed the provisioned throughput quotas of a table?	`ThrottledRequests` is incremented by one if any event within a request exceeds a provisioned throughput quota. Then, to gain insight into which event is throttling a request, compare `ThrottledRequests` with the `ReadThrottleEvents` and `WriteThrottleEvents` metrics for the table and its indexes.
How can I determine if any system errors occurred?	You can monitor `SystemErrors` to determine if any requests resulted in an HTTP 500 (server error) code. Typically, this metric should be equal to zero. If it isn't, then you might want to investigate.
How can I monitor the latency value for my table operations?	You can monitor `SuccessfulRequestLatency` by tracking the average latency and median latency through percentile metrics (p50). Occasional spikes in latency aren't a cause for concern. However, if average latency or p50 (median) is high, there could be an underlying issue that you must resolve. See Troubleshooting latency issues in Amazon DynamoDB for more information.

Viewing metrics in the CloudWatch console

Metrics are grouped by the service namespace first and then by the various dimension combinations within each namespace.

To view metrics in the CloudWatch console

1. Open the CloudWatch console at https://console.aws.amazon.com/cloudwatch/.
2. In the navigation pane, choose **Metrics, All metrics**.
3. Select the **DynamoDB** namespace. You can also select **Usage** namespace to view DynamoDB usage metrics. For more information about usage metrics, see AWS usage metrics.
4. The **Browse** tab displays all metrics in the namespace.
5. (Optional) To add the metric graph to a CloudWatch dashboard, choose **Actions, Add to dashboard**.

Viewing metrics in the AWS CLI

To obtain metric information using the AWS CLI, use the CloudWatch command `list-metrics`. In the following example, you list all metrics in the `AWS/DynamoDB` namespace.

```
aws cloudwatch list-metrics --namespace "AWS/DynamoDB"
```

To obtain metric statistics, use the command `get-metric-statistics`. The following command gets `ConsumedReadCapacityUnits` statistics for the table `ProductCatalog` over the specific 24-hour period, with a 5-minute granularity.

```
aws cloudwatch get-metric-statistics –namespace AWS/DynamoDB \
     –metric-name ConsumedReadCapacityUnits \
     –start-time 2023-11-01T00:00:00Z \
     –end-time 2023-11-02T00:00:00Z \
     –period 360 \
     –statistics Average \
     –dimensions Name=TableName,Value=ProductCatalog
```

Sample output appears as follows:

```
{
    "Datapoints": [
        {
            "Timestamp": "2023-11-01T 09:18:00+00:00",
```

```
            "Average": 20,
            "Unit": "Count"
        },
        {
            "Timestamp": "2023-11-01T 04:36:00+00:00",
            "Average": 22.5,
            "Unit": "Count"
        },
        {
            "Timestamp": "2023-11-01T 15:12:00+00:00",
            "Average": 20,
            "Unit": "Count"
        }, ...
        {
            "Timestamp": "2023-11-01T 17:30:00+00:00",
            "Average": 25,
            "Unit": "Count"
        }
    ],
    "Label": " ConsumedReadCapacityUnits "
}
```

DynamoDB Metrics and dimensions

When you interact with DynamoDB, it sends metrics and dimensions to CloudWatch.

DynamoDB outputs consumed provisioned throughput for one-minute periods. Auto scaling triggers when your consumed capacity breaches the configured target utilization for two consecutive minutes. CloudWatch alarms might have a short delay of up to a few minutes before triggering auto scaling. This delay ensures accurate CloudWatch metric evaluation. If the consumed throughput spikes are more than a minute apart, auto scaling might not trigger. Similarly, a scale down event can occur when 15 consecutive data points are lower than the target utilization. In either case, after auto scaling triggers, the UpdateTable API is invoked. It then takes several minutes to update the provisioned capacity for the table or index. During this period, any requests that exceed the previous provisioned capacity of the tables are throttled.

Viewing metrics and dimensions

CloudWatch displays the following metrics for DynamoDB:

DynamoDB metrics

Note

Amazon CloudWatch aggregates these metrics at one-minute intervals:

- ConditionalCheckFailedRequests
- ConsumedReadCapacityUnits
- ConsumedWriteCapacityUnits
- ReadAccountLimitThrottleEvents
- ReadKeyRangeThroughputThrottleEvents
- ReadMaxOnDemandThroughputThrottleEvents
- ReadProvisionedThroughputThrottleEvents
- ReadThrottleEvents
- ReturnedBytes
- ReturnedItemCount
- ReturnedRecordsCount
- SuccessfulRequestLatency
- SystemErrors
- TimeToLiveDeletedItemCount
- ThrottledRequests
- TransactionConflict
- UserErrors
- WriteAccountLimitThrottleEvents
- WriteKeyRangeThroughputThrottleEvents
- WriteMaxOnDemandThroughputThrottleEvents
- WriteProvisionedThroughputThrottleEvents
- WriteThrottleEvents
- FaultInjectionServiceInducedErrors

For all other DynamoDB metrics, the aggregation granularity is five minutes.

Not all statistics, such as *Average* or *Sum*, are applicable for every metric. However, all of these values are available through the Amazon DynamoDB console, or by using the CloudWatch console, AWS CLI, or AWS SDKs for all metrics.

In the following list, each metric has a set of valid statistics that are applicable to that metric.

List of Available Metrics

- AccountMaxReads
- AccountMaxTableLevelReads
- AccountMaxTableLevelWrites
- AccountMaxWrites
- AccountProvisionedReadCapacityUtilization
- AccountProvisionedWriteCapacityUtilization
- AgeOfOldestUnreplicatedRecord
- ConditionalCheckFailedRequests
- ConsumedChangeDataCaptureUnits
- ConsumedReadCapacityUnits
- ConsumedWriteCapacityUnits
- FailedToReplicateRecordCount
- MaxProvisionedTableReadCapacityUtilization
- MaxProvisionedTableWriteCapacityUtilization
- OnDemandMaxReadRequestUnits
- OnDemandMaxWriteRequestUnits
- OnlineIndexConsumedWriteCapacity
- OnlineIndexPercentageProgress
- OnlineIndexThrottleEvents
- PendingReplicationCount
- ProvisionedReadCapacityUnits
- ProvisionedWriteCapacityUnits
- ReadAccountLimitThrottleEvents
- ReadKeyRangeThroughputThrottleEvents

- ReadMaxOnDemandThroughputThrottleEvents
- ReadProvisionedThroughputThrottleEvents
- ReadThrottleEvents
- ReplicationLatency
- ReturnedBytes
- ReturnedItemCount
- ReturnedRecordsCount
- SuccessfulRequestLatency
- SystemErrors
- TimeToLiveDeletedItemCount
- ThrottledPutRecordCount
- ThrottledRequests
- TransactionConflict
- UserErrors
- WriteAccountLimitThrottleEvents
- WriteKeyRangeThroughputThrottleEvents
- WriteMaxOnDemandThroughputThrottleEvents
- WriteProvisionedThroughputThrottleEvents
- WriteThrottleEvents
- Usage metrics
- FaultInjectionServiceInducedErrors

AccountMaxReads

The maximum number of read capacity units that can be used by an account. This limit doesn't apply to on-demand tables or global secondary indexes.

Units: `Count`

Valid Statistics:

- `Maximum` – The maximum number of read capacity units that can be used by an account.

AccountMaxTableLevelReads

The maximum number of read capacity units that can be used by a table or global secondary index of an account. For on-demand tables, this limit caps the maximum read request units a table or a global secondary index can use.

Units: `Count`

Valid Statistics:

- `Maximum` – The maximum number of read capacity units that can be used by a table or global secondary index of the account.

AccountMaxTableLevelWrites

The maximum number of write capacity units that can be used by a table or global secondary index of an account. For on-demand tables, this limit caps the maximum write request units a table or a global secondary index can use.

Units: `Count`

Valid Statistics:

- `Maximum` – The maximum number of write capacity units that can be used by a table or global secondary index of the account.

AccountMaxWrites

The maximum number of write capacity units that can be used by an account. This limit doesn't apply to on-demand tables or global secondary indexes.

Units: `Count`

Valid Statistics:

- `Maximum` – The maximum number of write capacity units that can be used by an account.

AccountProvisionedReadCapacityUtilization

The percentage of provisioned read capacity units utilized by an account.

Units: `Percent`

Valid Statistics:

- `Maximum` – The maximum percentage of provisioned read capacity units utilized by the account.
- `Minimum` – The minimum percentage of provisioned read capacity units utilized by the account.
- `Average` – The average percentage of provisioned read capacity units utilized by the account. The metric is published for five-minute intervals. Therefore, if you rapidly adjust the provisioned read capacity units, this statistic might not reflect the true average.

AccountProvisionedWriteCapacityUtilization

The percentage of provisioned write capacity units utilized by an account.

Units: `Percent`

Valid Statistics:

- `Maximum` – The maximum percentage of provisioned write capacity units utilized by the account.
- `Minimum` – The minimum percentage of provisioned write capacity units utilized by the account.
- `Average` – The average percentage of provisioned write capacity units utilized by the account. The metric is published for five-minute intervals. Therefore, if you rapidly adjust the provisioned write capacity units, this statistic might not reflect the true average.

AgeOfOldestUnreplicatedRecord

The elapsed time since a record yet to be replicated to the Kinesis data stream first appeared in the DynamoDB table.

Units: `Milliseconds`

Dimensions: `TableName, DelegatedOperation`

Valid Statistics:

- `Maximum`.
- `Minimum`.
- `Average`.

ConditionalCheckFailedRequests

The number of failed attempts to perform conditional writes. The `PutItem`, `UpdateItem`, and `DeleteItem` operations let you provide a logical condition that must evaluate to true before the operation can proceed. If this condition evaluates to false, `ConditionalCheckFailedRequests` is incremented by one. `ConditionalCheckFailedRequests` is also incremented by one for PartiQL Update and Delete statements where a logical condition is provided and that condition evaluates to false.

> **Note**
>
> A failed conditional write will result in an HTTP 400 error (Bad Request). These events are reflected in the `ConditionalCheckFailedRequests` metric, but not in the `UserErrors` metric.

Units: `Count`

Dimensions: `TableName`

Valid Statistics:

- `Minimum`
- `Maximum`
- `Average`
- `SampleCount`
- `Sum`

ConsumedChangeDataCaptureUnits

The number of consumed change data capture units.

Units: `Count`

Dimensions: `TableName, DelegatedOperation`

Valid Statistics:

- `Minimum`
- `Maximum`
- `Average`

ConsumedReadCapacityUnits

The number of read capacity units consumed over the specified time period for both provisioned and on-demand capacity, so you can track how much of your throughput is used. You can retrieve the total consumed read capacity for a table and all of its global secondary indexes, or for a particular global secondary index. For more information, see Read/Write Capacity Mode.

The `TableName` dimension returns the `ConsumedReadCapacityUnits` for the table, but not for any global secondary indexes. To view `ConsumedReadCapacityUnits` for a global secondary index, you must specify both `TableName` and `GlobalSecondaryIndexName`.

> **Note**
>
> This means that short, intense spikes in capacity consumption lasting just a second may not be accurately reflected in the CloudWatch graph, potentially leading to a lower apparent consumption rate for that minute.
> Use the `Sum` statistic to calculate the consumed throughput. For example, get the `Sum` value over a span of one minute, and divide it by the number of seconds in a minute (60) to calculate the average `ConsumedReadCapacityUnits` per second. You can compare the calculated value to the provisioned throughput value that you provide DynamoDB.

Units: `Count`

Dimensions: `TableName, GlobalSecondaryIndexName`

Valid Statistics:

- `Minimum` – The minimum number of read capacity units consumed by any individual request to the table or index.
- `Maximum` – The maximum number of read capacity units consumed by any individual request to the table or index.
- `Average` – The average per-request read capacity consumed.

> **Note**
>
> The `Average` value is influenced by periods of inactivity where the sample value will be zero.

- `Sum` – The total read capacity units consumed. This is the most useful statistic for the `ConsumedReadCapacityUnits` metric.
- `SampleCount` – represents the frequency at which the metric is emitted. Even tables with zero traffic will have the `SampleCount` emitted regularly, but the sample values will always be zero.

> **Note**
>
> The `SampleCount` value is influenced by periods of inactivity where the sample value will be zero.

ConsumedWriteCapacityUnits

The number of write capacity units consumed over the specified time period for both provisioned and on-demand capacity, so you can track how much of your throughput is used. You can retrieve the total consumed write capacity for a table and all of its global secondary indexes, or for a particular global secondary index. For more information, see Read/Write Capacity Mode.

The `TableName` dimension returns the `ConsumedWriteCapacityUnits` for the table, but not for any global secondary indexes. To view `ConsumedWriteCapacityUnits` for a global secondary index, you must specify both `TableName` and `GlobalSecondaryIndexName`. The `Source` dimension can return either of two values: `Customer` and `GlobalTable`. Replicated writes will have `ConsumedWriteCapacityUnits` with the source `GlobalTable`, but regional table writes will have `ConsumedWriteCapacityUnits` with the source `Customer`.

> **Note**
>
> Use the `Sum` statistic to calculate the consumed throughput. For example, get the `Sum` value over a span of one minute, and divide it by the number of seconds in a minute (60) to calculate the average `ConsumedWriteCapacityUnits` per second (recognizing that this average doesn't highlight any large but brief spikes in write activity that occurred during

that minute). You can compare the calculated value to the provisioned throughput value that you provide DynamoDB.

Units: `Count`

Dimensions: `TableName, GlobalSecondaryIndexName, Source`

Valid Statistics:

- `Minimum` – The minimum number of write capacity units consumed by any individual request to the table or index.
- `Maximum` – The maximum number of write capacity units consumed by any individual request to the table or index.
- `Average` – The average per-request write capacity consumed.

> **Note**
>
> The `Average` value is influenced by periods of inactivity where the sample value will be zero.

- Sum – The total write capacity units consumed. This is the most useful statistic for the `ConsumedWriteCapacityUnits` metric.
- `SampleCount` – represents the frequency at which the metric is emitted. Even tables with zero traffic will have the `SampleCount` emitted regularly, but the sample values will always be zero.

> **Note**
>
> The `SampleCount` value is influenced by periods of inactivity where the sample value will be zero.

FailedToReplicateRecordCount

The number of records that DynamoDB failed to replicate to your Kinesis data stream.

Units: `Count`

Dimensions: `TableName`, `DelegatedOperation`

Valid Statistics:

- Sum

MaxProvisionedTableReadCapacityUtilization

The percentage of provisioned read capacity utilized by the highest provisioned read table or global secondary index of an account.

Units: Percent

Valid Statistics:

- Maximum – The maximum percentage of provisioned read capacity units utilized by the highest provisioned read table or global secondary index of an account.
- Minimum – The minimum percentage of provisioned read capacity units utilized by the highest provisioned read table or global secondary index of an account.
- Average – The average percentage of provisioned read capacity units utilized by the highest provisioned read table or global secondary index of the account. The metric is published for five-minute intervals. Therefore, if you rapidly adjust the provisioned read capacity units, this statistic might not reflect the true average.

MaxProvisionedTableWriteCapacityUtilization

The percentage of provisioned write capacity utilized by the highest provisioned write table or global secondary index of an account.

Units: Percent

Valid Statistics:

- Maximum – The maximum percentage of provisioned write capacity units utilized by the highest provisioned write table or global secondary index of an account.
- Minimum – The minimum percentage of provisioned write capacity units utilized by the highest provisioned write table or global secondary index of an account.
- Average – The average percentage of provisioned write capacity units utilized by the highest provisioned write table or global secondary index of the account. The metric is published for five-minute intervals. Therefore, if you rapidly adjust the provisioned write capacity units, this statistic might not reflect the true average.

OnDemandMaxReadRequestUnits

The number of specified on-demand read request units for a table or a global secondary index.

To view `OnDemandMaxReadRequestUnits` for a table, you must specify `TableName`. To view `OnDemandMaxReadRequestUnits` for a global secondary index, you must specify both `TableName` and `GlobalSecondaryIndexName`.

Units: Count

Dimensions: `TableName`, `GlobalSecondaryIndexName`

Valid Statistics:

- `Minimum` – The lowest setting for on-demand read request units. If you use `UpdateTable` to increase read request units, this metric shows the lowest value of on-demand `ReadRequestUnits` during this time period.
- `Maximum` – The highest setting for on-demand read request units. If you use `UpdateTable` to decrease read request units, this metric shows the highest value of on-demand `ReadRequestUnits` during this time period.
- `Average` – The average on-demand read request units. The `OnDemandMaxReadRequestUnits` metric is published at five-minute intervals. Therefore, if you rapidly adjust the on-demand read request units, this statistic might not reflect the true average.

OnDemandMaxWriteRequestUnits

The number of specified on-demand write request units for a table or a global secondary index.

To view `OnDemandMaxWriteRequestUnits` for a table, you must specify `TableName`. To view `OnDemandMaxWriteRequestUnits` for a global secondary index, you must specify both `TableName` and `GlobalSecondaryIndexName`.

Units: `Count`

Dimensions: `TableName`, `GlobalSecondaryIndexName`

Valid Statistics:

- `Minimum` – The lowest setting for on-demand write request units. If you use `UpdateTable` to increase write request units, this metric shows the lowest value of on-demand `WriteRequestUnits` during this time period.

- `Maximum` – The highest setting for on-demand write request units. If you use `UpdateTable` to decrease write request units, this metric shows the highest value of on-demand `WriteRequestUnits` during this time period.
- `Average` – The average on-demand write request units. The `OnDemandMaxWriteRequestUnits` metric is published at five-minute intervals. Therefore, if you rapidly adjust the on-demand write request units, this statistic might not reflect the true average.

OnlineIndexConsumedWriteCapacity

This metric is expected to show 0 during index builds. This metric previously showed the number of write capacity units consumed when adding a new global secondary index to a table.

Units: `Count`

Dimensions: `TableName, GlobalSecondaryIndexName`

Valid Statistics:

- `Minimum`
- `Maximum`
- `Average`
- `SampleCount`
- `Sum`

OnlineIndexPercentageProgress

The percentage of completion when a new global secondary index is being added to a table. DynamoDB must first allocate resources for the new index, and then backfill attributes from the table into the index. For large tables, this process might take a long time. You should monitor this statistic to view the relative progress as DynamoDB builds the index.

Units: `Count`

Dimensions: `TableName, GlobalSecondaryIndexName`

Valid Statistics:

- `Minimum`
- `Maximum`
- `Average`
- `SampleCount`
- `Sum`

OnlineIndexThrottleEvents

This metric is expected to show 0 during index builds. This metric previously showed the number of write throttle events that occur when adding a new global secondary index to a table.

Units: `Count`

Dimensions: `TableName, GlobalSecondaryIndexName`

Valid Statistics:

- `Minimum`
- `Maximum`
- `Average`
- `SampleCount`
- `Sum`

PendingReplicationCount

Metric for Global tables version 2017.11.29 (Legacy) (global tables only). The number of item updates that are written to one replica table, but that have not yet been written to another replica in the global table.

Units: `Count`

Dimensions: `TableName, ReceivingRegion`

Valid Statistics:

- `Average`
- `Sample Count`

- Sum

ProvisionedReadCapacityUnits

The number of provisioned read capacity units for a table or a global secondary index. The `TableName` dimension returns the `ProvisionedReadCapacityUnits` for the table, but not for any global secondary indexes. To view `ProvisionedReadCapacityUnits` for a global secondary index, you must specify both `TableName` and `GlobalSecondaryIndexName`.

Units:`Count`

Dimensions: `TableName`, `GlobalSecondaryIndexName`

Valid Statistics:

- `Minimum` – The lowest setting for provisioned read capacity. If you use `UpdateTable` to increase read capacity, this metric shows the lowest value of provisioned `ReadCapacityUnits` during this time period.
- `Maximum` – The highest setting for provisioned read capacity. If you use `UpdateTable` to decrease read capacity, this metric shows the highest value of provisioned `ReadCapacityUnits` during this time period.
- `Average` – The average provisioned read capacity. The `ProvisionedReadCapacityUnits` metric is published at five-minute intervals. Therefore, if you rapidly adjust the provisioned read capacity units, this statistic might not reflect the true average.

ProvisionedWriteCapacityUnits

The number of provisioned write capacity units for a table or a global secondary index.

The `TableName` dimension returns the `ProvisionedWriteCapacityUnits` for the table, but not for any global secondary indexes. To view `ProvisionedWriteCapacityUnits` for a global secondary index, you must specify both `TableName` and `GlobalSecondaryIndexName`.

Units: `Count`

Dimensions: `TableName`, `GlobalSecondaryIndexName`

Valid Statistics:

- `Minimum` – The lowest setting for provisioned write capacity. If you use `UpdateTable` to increase write capacity, this metric shows the lowest value of provisioned `WriteCapacityUnits` during this time period.
- `Maximum` – The highest setting for provisioned write capacity. If you use `UpdateTable` to decrease write capacity, this metric shows the highest value of provisioned `WriteCapacityUnits` during this time period.
- `Average` – The average provisioned write capacity. The `ProvisionedWriteCapacityUnits` metric is published at five-minute intervals. Therefore, if you rapidly adjust the provisioned write capacity units, this statistic might not reflect the true average.

ReadAccountLimitThrottleEvents

The number of read requests throttled due to account limits.

Units: `Count`

Dimensions: `TableName`, `GlobalSecondaryIndexName`

Valid Statistics:

- Sum – Total number of throttled events.
- `SampleCount` – Number of throttling occurrences.
- `Minimum` – Minimum number of throttled events in any given sample.
- `Maximum` – Maximum number of throttled events in any given sample.

ReadKeyRangeThroughputThrottleEvents

The number of read requests throttled due to partition limits.

Units: `Count`

Dimensions: `TableName`, `GlobalSecondaryIndexName`

Valid Statistics:

- Sum – Total number of throttled events.
- `SampleCount` – Number of throttling occurrences.
- `Minimum` – Minimum number of throttled events in any given sample.

- `Maximum` – Maximum number of throttled events in any given sample.

ReadMaxOnDemandThroughputThrottleEvents

The number of read requests throttled due to on-demand maximum throughput.

Units: `Count`

Dimensions: `TableName`, `GlobalSecondaryIndexName`

Valid Statistics:

- Sum – Total number of throttled events.
- `SampleCount` – Number of throttling occurrences.
- `Minimum` – Minimum number of throttled events in any given sample.
- `Maximum` – Maximum number of throttled events in any given sample.

ReadProvisionedThroughputThrottleEvents

The number of read requests throttled due to provisioned throughput limits.

Units: `Count`

Dimensions: `TableName`, `GlobalSecondaryIndexName`

Valid Statistics:

- Sum – Total number of throttled events.
- `SampleCount` – Number of throttling occurrences.
- `Minimum` – Minimum number of throttled events in any given sample.
- `Maximum` – Maximum number of throttled events in any given sample.

ReadThrottleEvents

Requests to DynamoDB that exceed the provisioned read capacity units for a table or a global secondary index.

A single request can result in multiple events. For example, a `BatchGetItem` that reads 10 items is processed as 10 `GetItem` events. For each event, `ReadThrottleEvents` is incremented by one

if that event is throttled. The ThrottledRequests metric for the entire BatchGetItem is not incremented unless *all 10* of the GetItem events are throttled.

The TableName dimension returns the ReadThrottleEvents for the table, but not for any global secondary indexes. To view ReadThrottleEvents for a global secondary index, you must specify both TableName and GlobalSecondaryIndexName.

Units: Count

Dimensions: TableName, GlobalSecondaryIndexName

Valid Statistics:

- SampleCount
- Sum

ReplicationLatency

(This metric is for DynamoDB global tables.) The elapsed time between an updated item appearing in the DynamoDB stream for one replica table, and that item appearing in another replica in the global table.

Units: Milliseconds

Dimensions: TableName, ReceivingRegion

Valid Statistics:

- Average
- Minimum
- Maximum

ReturnedBytes

The number of bytes returned by GetRecords operations (Amazon DynamoDB Streams) during the specified time period.

Units: Bytes

Dimensions: Operation, StreamLabel, TableName

Valid Statistics:

- Minimum
- Maximum
- Average
- SampleCount
- Sum

ReturnedItemCount

The number of items returned by Query, Scan or ExecuteStatement (select) operations during the specified time period.

The number of items *returned* is not necessarily the same as the number of items that were evaluated. For example, suppose that you requested a Scan on a table or an index that had 100 items, but specified a FilterExpression that narrowed the results so that only 15 items were returned. In this case, the response from Scan would contain a ScanCount of 100 and a Count of 15 returned items.

Units: Count

Dimensions: TableName, Operation

Valid Statistics:

- Minimum
- Maximum
- Average
- SampleCount
- Sum

ReturnedRecordsCount

The number of stream records returned by GetRecords operations (Amazon DynamoDB Streams) during the specified time period.

Units: Count

Dimensions: `Operation, StreamLabel, TableName`

Valid Statistics:

- `Minimum`
- `Maximum`
- `Average`
- `SampleCount`
- `Sum`

SuccessfulRequestLatency

The latency of successful requests to DynamoDB or Amazon DynamoDB Streams during the specified time period. `SuccessfulRequestLatency` can provide two different kinds of information:

- The elapsed time for successful requests (`Minimum`, `Maximum`, `Sum`, `Average`, or `Percentile`).
- The number of successful requests (`SampleCount`).

`SuccessfulRequestLatency` reflects activity only within DynamoDB or Amazon DynamoDB Streams, and doesn't consider network latency or client-side activity.

> **Note**
>
> To analyze custom percentile values (such as p99.9), you can manually enter the desired percentile (e.g., p99.9) in the CloudWatch metric statistic field. This allows you to evaluate latency distributions beyond the default percentiles listed in the dropdown.

Units: `Milliseconds`

Dimensions: `TableName, Operation, StreamLabel`

Valid Statistics:

- `Minimum`
- `Maximum`

- `Sum`
- `Average`
- `Percentile`
- `SampleCount`

SystemErrors

The requests to DynamoDB or Amazon DynamoDB Streams that generate an HTTP 500 status code during the specified time period. An HTTP 500 usually indicates an internal service error.

> **Note**
>
> When DynamoDB returns a system error (HTTP 500), most AWS SDKs automatically perform a configurable number of retries. If the issue resolves during a retry, your application continues without seeing the error, and you may notice increased client-side perceived latency. If the error persists after all retries, it propagates to your application code.

Units: `Count`

Dimensions: `TableName, Operation`

Valid Statistics:

- `Sum`
- `SampleCount`

TimeToLiveDeletedItemCount

The number of items deleted by Time to Live (TTL) during the specified time period. This metric helps you monitor the rate of TTL deletions on your table.

Units: `Count`

Dimensions: TableName

Valid Statistics:

- Sum

ThrottledPutRecordCount

The number of records that were throttled by your Kinesis data stream due to insufficient Kinesis Data Streams capacity.

Units: `Count`

Dimensions: TableName, DelegatedOperation

Valid Statistics:

- `Minimum`
- `Maximum`
- `Average`
- `SampleCount`

ThrottledRequests

Requests to DynamoDB that exceed the provisioned throughput limits on a resource (such as a table or an index).

`ThrottledRequests` is incremented by one if any event within a request exceeds a provisioned throughput limit. For example, if you update an item in a table with global secondary indexes, there are multiple events—a write to the table, and a write to each index. If one or more of these events are throttled, then `ThrottledRequests` is incremented by one.

> **Note**
>
> In a batch request (`BatchGetItem` or `BatchWriteItem`), `ThrottledRequests` is incremented only if *every* request in the batch is throttled.
> If any individual request within the batch is throttled, one of the following metrics is incremented:
>
> - `ReadThrottleEvents` – For a throttled `GetItem` event within `BatchGetItem`.
> - `WriteThrottleEvents` – For a throttled `PutItem` or `DeleteItem` event within `BatchWriteItem`.

To gain insight into which event is throttling a request, compare `ThrottledRequests` with the `ReadThrottleEvents` and `WriteThrottleEvents` for the table and its indexes.

> **Note**
>
> A throttled request will result in an HTTP 400 status code. All such events are reflected in the `ThrottledRequests` metric, but not in the `UserErrors` metric.

Units: `Count`

Dimensions: `TableName, Operation`

Valid Statistics:

- `Sum`
- `SampleCount`

TransactionConflict

Rejected item-level requests due to transactional conflicts between concurrent requests on the same items. For more information, see Transaction Conflict Handling in DynamoDB.

Units: `Count`

Dimensions: `TableName`

Valid Statistics:

- `Sum` – The number of rejected item-level requests due to transaction conflicts.

> **Note**
>
> If multiple item-level requests within a call to `TransactWriteItems` or `TransactGetItems` were rejected, `Sum` is incremented by one for each item-level `Put`, `Update`, `Delete`, or `Get` request.

- `SampleCount` – The number of rejected requests due to transaction conflicts.

> **Note**
>
> If multiple item-level requests within a call to `TransactWriteItems` or `TransactGetItems` are rejected, `SampleCount` is only incremented by one.

- `Min` – The minimum number of rejected item-level requests within a call to `TransactWriteItems`, `TransactGetItems`, `PutItem`, `UpdateItem`, or `DeleteItem`.
- `Max` – The maximum number of rejected item-level requests within a call to `TransactWriteItems`, `TransactGetItems`, `PutItem`, `UpdateItem`, or `DeleteItem`.
- `Average` – The average number of rejected item-level requests within a call to `TransactWriteItems`, `TransactGetItems`, `PutItem`, `UpdateItem`, or `DeleteItem`.

UserErrors

Requests to DynamoDB or Amazon DynamoDB Streams that generate an HTTP 400 status code during the specified time period. An HTTP 400 usually indicates a client-side error, such as an invalid combination of parameters, an attempt to update a nonexistent table, or an incorrect request signature.

Some examples of exceptions that will log metrics related to `UserErrors` would be:

- `ResourceNotFoundException`
- `ValidationException`
- `TransactionConflict`

All such events are reflected in the `UserErrors` metric, except for the following:

- *ProvisionedThroughputExceededException* – See the `ThrottledRequests` metric in this section.
- *ConditionalCheckFailedException* – See the `ConditionalCheckFailedRequests` metric in this section.

`UserErrors` represents the aggregate of HTTP 400 errors for DynamoDB or Amazon DynamoDB Streams requests for the current AWS Region and the current AWS account.

Units: `Count`

Valid Statistics:

- Sum
- SampleCount

WriteAccountLimitThrottleEvents

The number of write requests throttled due to account limits.

Units: Count

Dimensions: TableName

Valid Statistics:

- Sum – Total number of throttled events.
- SampleCount – Number of throttling occurrences.
- Minimum – Minimum number of throttled events in any given sample.
- Maximum – Maximum number of throttled events in any given sample.

WriteKeyRangeThroughputThrottleEvents

The number of write requests throttled due to partition limits.

Units: Count

Dimensions: TableName, GlobalSecondaryIndexName

Valid Statistics:

- Sum – Total number of throttled events.
- SampleCount – Number of throttling occurrences.
- Minimum – Minimum number of throttled events in any given sample.
- Maximum – Maximum number of throttled events in any given sample.

WriteMaxOnDemandThroughputThrottleEvents

The number of write requests throttled due to on-demand maximum throughput.

Units: Count

Dimensions: TableName, GlobalSecondaryIndexName

Valid Statistics:

- Sum – Total number of throttled events.
- SampleCount – Number of throttling occurrences.
- Minimum – Minimum number of throttled events in any given sample.
- Maximum – Maximum number of throttled events in any given sample.

WriteProvisionedThroughputThrottleEvents

The number of write requests throttled due to provisioned throughput limits.

Units: Count

Dimensions: TableName, GlobalSecondaryIndexName

Valid Statistics:

- Sum – Total number of throttled events.
- SampleCount – Number of throttling occurrences.
- Minimum – Minimum number of throttled events in any given sample.
- Maximum – Maximum number of throttled events in any given sample.

WriteThrottleEvents

Requests to DynamoDB that exceed the provisioned write capacity units for a table or a global secondary index.

A single request can result in multiple events. For example, a PutItem request on a table with three global secondary indexes would result in four events—the table write, and each of the three index writes. For each event, the WriteThrottleEvents metric is incremented by one if that event is throttled. For single PutItem requests, if any of the events are throttled, ThrottledRequests is also incremented by one. For BatchWriteItem, the ThrottledRequests metric for the entire BatchWriteItem is not incremented unless all of the individual PutItem or DeleteItem events are throttled.

The TableName dimension returns the WriteThrottleEvents for the table, but not for any global secondary indexes. To view WriteThrottleEvents for a global secondary index, you must specify both TableName and GlobalSecondaryIndexName.

Units: Count

Dimensions: TableName, GlobalSecondaryIndexName

Valid Statistics:

- Sum
- SampleCount

Usage metrics

Usage metrics in CloudWatch allow you to proactively manage usage by visualizing metrics in the CloudWatch console, creating custom dashboards, detecting changes in activity with CloudWatch anomaly detection, and configuring alarms that alert you when usage approaches a threshold.

DynamoDB also integrates these usage metrics with Service Quotas. You can use CloudWatch to manage your account's use of your service quotas. For more information, see Visualizing your service quotas and setting alarms

List of Available Usage Metrics

- AccountProvisionedWriteCapacityUnits
- AccountProvisionedReadCapacityUnits
- TableCount

AccountProvisionedWriteCapacityUnits

The sum of write capacity units provisioned for all tables and global secondary indexes of an account.

Units: Count

Valid Statistics:

- Minimum – The lowest number of provisioned write capacity units during a time period.

- `Maximum` – The highest number of provisioned write capacity units during a time period.
- `Average` - The average number provisioned write capacity units account during a time period.

This metric is published at five-minute intervals. Therefore, if you rapidly adjust the provisioned write capacity units, this statistic might not reflect the true average.

AccountProvisionedReadCapacityUnits

The sum of read capacity units provisioned for all tables and global secondary indexes of an account.

Units: `Count`

Valid Statistics:

- `Minimum` – The lowest number of provisioned read capacity units during a time period.
- `Maximum` – The highest number of provisioned read capacity units during a time period.
- `Average` - The average number provisioned read capacity units account during a time period.

This metric is published at five-minute intervals. Therefore, if you rapidly adjust the provisioned read capacity units, this statistic might not reflect the true average.

TableCount

The number of active tables of an account.

Units: `Count`

Valid Statistics:

- `Minimum` – The lowest number of tables during a time period.
- `Maximum` – The highest number of tables during a time period.
- `Average` - The average number tables during a time period.

FaultInjectionServiceInducedErrors

The requests to DynamoDB that generate a simulated HTTP 500 status code during the specified time period and during the catchup as a result of AWS FIS experiment.

Units: `Count`

Dimensions: `TableName`, `Operation`

Valid Statistics:

- `Sum`
- `SampleCount`

Understanding metrics and dimensions for DynamoDB

The metrics for DynamoDB are qualified by the values for the account, table name, global secondary index name, or operation. You can use the CloudWatch console to retrieve DynamoDB data along any of the dimensions in the table below.

List of Available Dimensions

- DelegatedOperation
- GlobalSecondaryIndexName
- Operation
- OperationType
- Verb
- ReceivingRegion
- StreamLabel
- TableName

DelegatedOperation

This dimension limits the data to operations DynamoDB performs on your behalf. The following operations are captured:

- Change data capture for Kinesis Data Streams.

GlobalSecondaryIndexName

This dimension limits the data to a global secondary index on a table. If you specify `GlobalSecondaryIndexName`, you must also specify `TableName`.

Operation

This dimension limits the data to one of the following DynamoDB operations:

- `PutItem`
- `DeleteItem`
- `UpdateItem`
- `GetItem`
- `BatchGetItem`
- `Scan`
- `Query`
- `BatchWriteItem`
- `TransactWriteItems`
- `TransactGetItems`
- `ExecuteTransaction`
- `BatchExecuteStatement`
- `ExecuteStatement`

In addition, you can limit the data to the following Amazon DynamoDB Streams operation:

- `GetRecords`

OperationType

This dimension limits the data to one of the following operation types:

- `Read`
- `Write`

This dimension is emitted for `ExecuteTransaction` and `BatchExecuteStatement` requests.

Verb

This dimension limits the data to one of the following DynamoDB PartiQL verbs:

- Insert: `PartiQLInsert`
- Select: `PartiQLSelect`
- Update: `PartiQLUpdate`
- Delete: `PartiQLDelete`

This dimension is emitted for the `ExecuteStatement` operation.

ReceivingRegion

This dimension limits the data to a particular AWS region. It is used with metrics originating from replica tables within a DynamoDB global table.

StreamLabel

This dimension limits the data to a specific stream label. It is used with metrics originating from Amazon DynamoDB Streams `GetRecords` operations.

TableName

This dimension limits the data to a specific table. This value can be any table name in the current region and the current AWS account.

Creating CloudWatch alarms in DynamoDB

A CloudWatch alarm watches a single metric over a specified time period, and performs one or more specified actions, based on the value of the metric relative to a threshold over time. The action is a notification sent to an Amazon SNS topic or Auto Scaling policy. You can also add alarms to dashboards so you can monitor and receive alerts about your AWS resources and applications across multiple regions. There is no limit to the number of alarms you can create. CloudWatch alarms do not invoke actions simply because they are in a particular state; the state must have changed and been maintained for a specified number of periods. For a list of recommended DynamoDB alarms, see Recommended alarms.

> **Note**
>
> You must specify all the required dimensions when creating your CloudWatch alarm, since CloudWatch will not aggregate metrics for a missing dimension. Creating a CloudWatch alarm with a missing dimension will not result in an error, when creating the alarm.

Assume you have a provisioned table with five read capacity units. You want to be notified before you consume the entire provisioned read capacity, so you decide to create a CloudWatch alarm to get notified when the consumed capacity reaches 80% of what you have provisioned for the table. You can create alarms in the CloudWatch console or using the AWS CLI.

Creating an alarm in the CloudWatch console

To create an alarm in the CloudWatch console

1. Sign in to the AWS Management Console and open the CloudWatch console at https://console.aws.amazon.com/cloudwatch/.
2. In the navigation pane, choose **Alarms**, **All alarms**.
3. Choose **Create alarm**.
4. Find the row with the table that you want to monitor and **`ConsumeReadCapacityUnits`** in the **Metric Name** column. Select the check box next to this row and choose **Select metric**.
5. Under **Specify metric and conditions**, for **Statistic** choose **Sum**. Choose a **Period** of **1 minute**.
6. Under **Conditions**, specify the following:
   a. For **Threshold type**, choose **Static**.
   b. For **Whenever `ConsumedReadCapacityUnits` is**, choose **Greater/Equal** and specify the threshold as 240.
7. Choose **Next**.
8. Under **Notification**, choose **`In alarm`** and select an SNS topic to notify when the alarm is in ALARM state.
9. When finished, choose **Next**.
10. Enter a name and description for the alarm and choose **Next**.
11. Under **Preview and create**, confirm that the information and conditions are what you want, then choose **Create alarm**.

Creating an alarm in the AWS CLI

```
aws cloudwatch put-metric-alarm \
    -\-alarm-name ReadCapacityUnitsLimitAlarm \
    -\-alarm-description "Alarm when read capacity reaches 80% of my provisioned read
 capacity" \
    -\-namespace AWS/DynamoDB \
```

```
    -\-metric-name ConsumedReadCapacityUnits \
    -\-dimensions Name=TableName,Value=myTable \
    -\-statistic Sum \
    -\-threshold 240 \
    -\-comparison-operator GreaterThanOrEqualToThreshold \
    -\-period 60 \
    -\-evaluation-periods 1 \
    -\-alarm-actions arn:aws:sns:us-east-1:123456789012:capacity-alarm
```

Test the alarm.

```
aws cloudwatch set-alarm-state -\-alarm-name ReadCapacityUnitsLimitAlarm -\-state-
reason "initializing" -\-state-value OK
```

```
aws cloudwatch set-alarm-state -\-alarm-name ReadCapacityUnitsLimitAlarm -\-state-
reason "initializing" -\-state-value ALARM
```

More AWS CLI examples

The following procedure describes how you're notified if you have requests that exceed the provisioned througput quotas of a table.

1. Create an Amazon SNS topic `arn:aws:sns:us-east-1:123456789012:requests-exceeding-throughput`. For more information, see Set up Amazon Simple Notification Service.
2. Create the alarm.

```
aws cloudwatch put-metric-alarm \
        -\-alarm-name ReadCapacityUnitsLimitAlarm \
        -\-alarm-description "Alarm when read capacity reaches 80% of my
 provisioned read capacity" \
        -\-namespace AWS/DynamoDB \
        -\-metric-name ConsumedReadCapacityUnits \
        -\-dimensions Name=TableName,Value=myTable \
        -\-statistic Sum \
        -\-threshold 240 \
        -\-comparison-operator GreaterThanOrEqualToThreshold \
        -\-period 60 \
        -\-evaluation-periods 1 \
        -\-alarm-actions arn:aws:sns:us-east-1:123456789012:capacity-alarm
```

3. Test the alarm.

```
aws cloudwatch set-alarm-state --alarm-name RequestsExceedingThroughputAlarm --
state-reason "initializing" --state-value OK
```

```
aws cloudwatch set-alarm-state --alarm-name RequestsExceedingThroughputAlarm --
state-reason "initializing" --state-value ALARM
```

The following procedure describes how you're notified if you get system errors.

1. Create an Amazon SNS topic `arn:aws:sns:us-east-1:123456789012:notify-on-system-errors`. For more information, see Set up Amazon Simple Notification Service.
2. Create the alarm.

```
aws cloudwatch put-metric-alarm \
    --alarm-name SystemErrorsAlarm \
    --alarm-description "Alarm when system errors occur" \
    --namespace AWS/DynamoDB \
    --metric-name SystemErrors \
    --dimensions Name=TableName,Value=myTable
 Name=Operation,Value=aDynamoDBOperation \
    --statistic Sum \
    --threshold 0 \
    --comparison-operator GreaterThanThreshold \
    --period 60 \
    --unit Count \
    --evaluation-periods 1 \
    --treat-missing-data breaching \
    --alarm-actions arn:aws:sns:us-east-1:123456789012:notify-on-system-errors
```

3. Test the alarm.

```
aws cloudwatch set-alarm-state --alarm-name SystemErrorsAlarm --state-reason
 "initializing" --state-value OK
```

```
aws cloudwatch set-alarm-state --alarm-name SystemErrorsAlarm --state-reason
 "initializing" --state-value ALARM
```

Logging DynamoDB operations by using AWS CloudTrail

DynamoDB is integrated with AWS CloudTrail, a service that provides a record of actions taken by a user, role, or an AWS service in DynamoDB. CloudTrail captures all API calls for DynamoDB as events. The calls captured include calls from the DynamoDB console and code calls to the DynamoDB API operations, using both PartiQL and the classic API. If you create a trail, you can enable continuous delivery of CloudTrail events to an Amazon S3 bucket, including events for DynamoDB. If you don't configure a trail, you can still view the most recent events in the CloudTrail console in **Event history**. Using the information collected by CloudTrail, you can determine the request that was made to DynamoDB, the IP address from which the request was made, who made the request, when it was made, and additional details.

For robust monitoring and alerting, you can also integrate CloudTrail events with Amazon CloudWatch Logs. To enhance your analysis of DynamoDB service activity and identify changes in activities for an AWS account, you can query AWS CloudTrail logs using Amazon Athena. For example, you can use queries to identify trends and further isolate activity by attributes such as source IP address or user.

To learn more about CloudTrail, including how to configure and enable it, see the AWS CloudTrail User Guide.

Topics

- DynamoDB information in CloudTrail
- Understanding DynamoDB log file entries

DynamoDB information in CloudTrail

CloudTrail is enabled on your AWS account when you create the account. When supported event activity occurs in DynamoDB, that activity is recorded in a CloudTrail event along with other AWS service events in **Event history**. You can view, search, and download recent events in your AWS account. For more information, see Working with CloudTrail Event history.

For an ongoing record of events in your AWS account, including events for DynamoDB, create a trail. A *trail* enables CloudTrail to deliver log files to an Amazon S3 bucket. By default, when you create a trail in the console, the trail applies to all AWS Regions. The trail logs events from all Regions in the AWS partition and delivers the log files to the Amazon S3 bucket that you specify. Additionally, you can configure other AWS services to further analyze and act upon the event data collected in CloudTrail logs. For more information, see the following:

- Overview for creating a trail
- CloudTrail supported services and integrations
- Configuring Amazon SNS notifications for CloudTrail
- Receiving CloudTrail log files from multiple regions and Receiving CloudTrail log files from multiple accounts

Control plane events in CloudTrail

The following API actions are logged by default as events in CloudTrail files:

Amazon DynamoDB

- CreateBackup
- CreateGlobalTable
- CreateTable
- DeleteBackup
- DeleteTable
- DescribeBackup
- DescribeContinuousBackups
- DescribeGlobalTable
- DescribeLimits
- DescribeTable
- DescribeTimeToLive
- ListBackups
- ListTables
- ListTagsOfResource
- ListGlobalTables
- RestoreTableFromBackup
- RestoreTableToPointInTime
- TagResource
- UntagResource
- UpdateGlobalTable
- UpdateTable

- UpdateTimeToLive
- DescribeReservedCapacity
- DescribeReservedCapacityOfferings
- PurchaseReservedCapacityOfferings
- DescribeScalableTargets
- RegisterScalableTarget

DynamoDB Streams

- DescribeStream
- ListStreams

DynamoDB Accelerator (DAX)

- CreateCluster
- CreateParameterGroup
- CreateSubnetGroup
- DecreaseReplicationFactor
- DeleteCluster
- DeleteParameterGroup
- DeleteSubnetGroup
- DescribeClusters
- DescribeDefaultParameters
- DescribeEvents
- DescribeParameterGroups
- DescribeParameters
- DescribeSubnetGroups
- IncreaseReplicationFactor
- ListTags
- RebootNode
- TagResource
- UntagResource

- UpdateCluster
- UpdateParameterGroup
- UpdateSubnetGroup

DynamoDB data plane events in CloudTrail

To enable logging of the following API actions in CloudTrail files, you'll need to enable logging of data plane API activity in CloudTrail. See Logging data events for trails for more information.

Data plane events can be filtered by resource type, for granular control over which DynamoDB API calls you want to selectively log and pay for in CloudTrail. For example, by specifying `AWS::DynamoDB::Stream` as a resource type, you can log only calls to the DynamoDB streams APIs. For tables with streams enabled, the resource field in the data plane event contains both `AWS::DynamoDB::Stream` and `AWS::DynamoDB::Table`. If you specify `AWS::DynamoDB::Table` as a resource type, it will log both DynamoDB table and DynamoDB streams events by default. You can add an additional filter to exclude the streams events, if you don't want the streams events to be logged. For more information, see DataResource in the *AWS CloudTrail API Reference*.

Amazon DynamoDB

- BatchExecuteStatement
- BatchGetItem
- BatchWriteItem
- DeleteItem
- ExecuteStatement
- ExecuteTransaction
- GetItem
- PutItem
- Query
- Scan
- TransactGetItems
- TransactWriteItems
- UpdateItem

> **Note**
>
> DynamoDB Time to Live data plane actions are not logged by CloudTrail

DynamoDB Streams

- GetRecords
- GetShardIterator

> **Important**
>
> When you log `GetRecords` data events, you might see `GetRecords` calls from DynamoDB internal operations, such as global tables replication. Although you are not charged by DynamoDB for these `GetRecords` calls, you are charged by CloudTrail for the data event logs. This might result in unexpected CloudTrail charges.
> To avoid unexpected CloudTrail charges, do one of the following:
>
> - Use the **Exclude AWS service-initiated events** log selector template.
> - Add an advanced event selector filter with `userIdentity.arn` set to `NotStartsWith` `AWSServiceRoleFor`.
>
> For more information, see Logging data events in the AWS CloudTrail User Guide.

Understanding DynamoDB log file entries

A trail is a configuration that enables delivery of events as log files to an Amazon S3 bucket that you specify. CloudTrail log files contain one or more log entries. An event represents a single request from any source and includes information about the requested action, the date and time of the action, request parameters, and so on.

Every event or log entry contains information about who generated the request. The identity information helps you determine the following:

- Whether the request was made with root or user credentials.
- Whether the request was made with temporary security credentials for a role or federated user.

- Whether the request was made by another AWS service.

> **Note**
>
> Non key attribute values will be redacted in the CloudTrail logs of actions using the PartiQL API, and will not appear in logs of actions using the classic API.

For more information, see the CloudTrail userIdentity element.

The following examples demonstrate CloudTrail logs of these event types:

Amazon DynamoDB

- `UpdateTable`
- `DeleteTable`
- `CreateCluster`
- `PutItem` (Successful)
- `UpdateItem` (Unsuccessful)
- `TransactWriteItems` (Successful)
- `TransactWriteItems` (With `TransactionCanceledException`)
- `ExecuteStatement`
- `BatchExecuteStatement`

DynamoDB Streams

- `GetRecords`

`UpdateTable`

```
{
    "Records": [
        {
            "eventVersion": "1.03",
            "userIdentity": {
                "type": "AssumedRole",
                "principalId": "AKIAIOSFODNN7EXAMPLE:bob",
```

```
                "arn": "arn:aws:sts::111122223333:assumed-role/users/bob",
                "accountId": "111122223333",
                "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
                "sessionContext": {
                    "attributes": {
                        "mfaAuthenticated": "false",
                        "creationDate": "2015-05-28T18:06:01Z"
                    },
                    "sessionIssuer": {
                        "type": "Role",
                        "principalId": "AKIAI44QH8DHBEXAMPLE",
                        "arn": "arn:aws:iam::444455556666:role/admin-role",
                        "accountId": "444455556666",
                        "userName": "bob"
                    }
                }
            },
            "eventTime": "2015-05-04T02:14:52Z",
            "eventSource": "dynamodb.amazonaws.com",
            "eventName": "UpdateTable",
            "awsRegion": "us-west-2",
            "sourceIPAddress": "192.0.2.0",
            "userAgent": "console.aws.amazon.com",
            "requestParameters": {
                "provisionedThroughput": {
                    "writeCapacityUnits": 25,
                    "readCapacityUnits": 25
                }
            },
            "responseElements": {
                "tableDescription": {
                    "tableName": "Music",
                    "attributeDefinitions": [
                        {
                            "attributeType": "S",
                            "attributeName": "Artist"
                        },
                        {
                            "attributeType": "S",
                            "attributeName": "SongTitle"
                        }
                    ],
                    "itemCount": 0,
                    "provisionedThroughput": {
```

```
                        "writeCapacityUnits": 10,
                        "numberOfDecreasesToday": 0,
                        "readCapacityUnits": 10,
                        "lastIncreaseDateTime": "May 3, 2015 11:34:14 PM"
                    },
                    "creationDateTime": "May 3, 2015 11:34:14 PM",
                    "keySchema": [
                        {
                            "attributeName": "Artist",
                            "keyType": "HASH"
                        },
                        {
                            "attributeName": "SongTitle",
                            "keyType": "RANGE"
                        }
                    ],
                    "tableStatus": "UPDATING",
                    "tableSizeBytes": 0
                }
            },
            "requestID": "AALNP0J2L244N5O15PKISJ1KUFVV4KQNSO5AEMVJF66Q9ASUAAJG",
            "eventID": "eb834e01-f168-435f-92c0-c36278378b6e",
            "eventType": "AwsApiCall",
            "apiVersion": "2012-08-10",
            "recipientAccountId": "111122223333"
        }
    ]
}
```

DeleteTable

```
{
    "Records": [
        {
            "eventVersion": "1.03",
            "userIdentity": {
                "type": "AssumedRole",
                "principalId": "AKIAIOSFODNN7EXAMPLE:bob",
                "arn": "arn:aws:sts::111122223333:assumed-role/users/bob",
                "accountId": "111122223333",
                "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
                "sessionContext": {
                    "attributes": {
```

```
                        "mfaAuthenticated": "false",
                        "creationDate": "2015-05-28T18:06:01Z"
                    },
                    "sessionIssuer": {
                        "type": "Role",
                        "principalId": "AKIAI44QH8DHBEXAMPLE",
                        "arn": "arn:aws:iam::444455556666:role/admin-role",
                        "accountId": "444455556666",
                        "userName": "bob"
                    }
                }
            },
            "eventTime": "2015-05-04T13:38:20Z",
            "eventSource": "dynamodb.amazonaws.com",
            "eventName": "DeleteTable",
            "awsRegion": "us-west-2",
            "sourceIPAddress": "192.0.2.0",
            "userAgent": "console.aws.amazon.com",
            "requestParameters": {
                "tableName": "Music"
            },
            "responseElements": {
                "tableDescription": {
                    "tableName": "Music",
                    "itemCount": 0,
                    "provisionedThroughput": {
                        "writeCapacityUnits": 25,
                        "numberOfDecreasesToday": 0,
                        "readCapacityUnits": 25
                    },
                    "tableStatus": "DELETING",
                    "tableSizeBytes": 0
                }
            },
            "requestID": "4KBNVRGD25RG1KEO9UT4V3FQDJVV4KQNSO5AEMVJF66Q9ASUAAJG",
            "eventID": "a954451c-c2fc-4561-8aea-7a30ba1fdf52",
            "eventType": "AwsApiCall",
            "apiVersion": "2012-08-10",
            "recipientAccountId": "111122223333"
        }
    ]
}
```

CreateCluster


```
{
    "Records": [
        {
            "eventVersion": "1.05",
            "userIdentity": {
                "type": "IAMUser",
                "principalId": "AKIAIOSFODNN7EXAMPLE:bob",
                "arn": "arn:aws:sts::111122223333:assumed-role/users/bob",
                "accountId": "111122223333",
                "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
                "userName": "bob"
            },
            "eventTime": "2019-12-17T23:17:34Z",
            "eventSource": "dax.amazonaws.com",
            "eventName": "CreateCluster",
            "awsRegion": "us-west-2",
            "sourceIPAddress": "192.0.2.0",
            "userAgent": "aws-cli/1.16.304 Python/3.6.9
 Linux/4.9.184-0.1.ac.235.83.329.metal1.x86_64 botocore/1.13.40",
            "requestParameters": {
                "sSESpecification": {
                    "enabled": true
                },
                "clusterName": "daxcluster",
                "nodeType": "dax.r4.large",
                "replicationFactor": 3,
                "iamRoleArn": "arn:aws:iam::111122223333:role/
DAXServiceRoleForDynamoDBAccess"
            },
            "responseElements": {
                "cluster": {
                    "securityGroups": [
                        {
                            "securityGroupIdentifier": "sg-1af6e36e",
                            "status": "active"
                        }
                    ],
                    "parameterGroup": {
                        "nodeIdsToReboot": [],
                        "parameterGroupName": "default.dax1.0",
                        "parameterApplyStatus": "in-sync"
```

```
                    },
                    "clusterDiscoveryEndpoint": {
                        "port": 8111
                    },
                    "clusterArn": "arn:aws:dax:us-west-2:111122223333:cache/
daxcluster",
                    "status": "creating",
                    "subnetGroup": "default",
                    "sSEDescription": {
                        "status": "ENABLED",
                        "kMSMasterKeyArn": "arn:aws:kms:us-
west-2:111122223333:key/764898e4-adb1-46d6-a762-e2f4225b4fc4"
                    },
                    "iamRoleArn": "arn:aws:iam::111122223333:role/
DAXServiceRoleForDynamoDBAccess",
                    "clusterName": "daxcluster",
                    "activeNodes": 0,
                    "totalNodes": 3,
                    "preferredMaintenanceWindow": "thu:13:00-thu:14:00",
                    "nodeType": "dax.r4.large"
                }
            },
            "requestID": "585adc5f-ad05-4e27-8804-70ba1315f8fd",
            "eventID": "29158945-28da-4e32-88e1-56d1b90c1a0c",
            "eventType": "AwsApiCall",
            "recipientAccountId": "111122223333"
        }
    ]
}
```

`PutItem` (Successful)

```
{
    "Records": [
        {
            "eventVersion": "1.06",
            "userIdentity": {
                "type": "AssumedRole",
                "principalId": "AKIAIOSFODNN7EXAMPLE:bob",
                "arn": "arn:aws:sts::111122223333:assumed-role/users/bob",
                "accountId": "111122223333",
                "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
                "sessionContext": {
```

```
                    "attributes": {
                        "mfaAuthenticated": "false",
                        "creationDate": "2015-05-28T18:06:01Z"
                    },
                    "sessionIssuer": {
                        "type": "Role",
                        "principalId": "AKIAI44QH8DHBEXAMPLE",
                        "arn": "arn:aws:iam::444455556666:role/admin-role",
                        "accountId": "444455556666",
                        "userName": "bob"
                    }
                }
            },
            "eventTime": "2019-01-19T15:41:54Z",
            "eventSource": "dynamodb.amazonaws.com",
            "eventName": "PutItem",
            "awsRegion": "us-west-2",
            "sourceIPAddress": "192.0.2.0",
            "userAgent": "aws-cli/1.15.64 Python/2.7.16 Darwin/17.7.0
botocore/1.10.63",
            "requestParameters": {
                "tableName": "Music",
                "key": {
                    "Artist": "No One You Know",
                    "SongTitle": "Scared of My Shadow"
                },
                "item": [
                    "Artist",
                    "SongTitle",
                    "AlbumTitle"
                ],
                "returnConsumedCapacity": "TOTAL"
            },
            "responseElements": null,
            "requestID": "4KBNVRGD25RG1KEO9UT4V3FQDJVV4KQNSO5AEMVJF66Q9ASUAAJG",
            "eventID": "a954451c-c2fc-4561-8aea-7a30ba1fdf52",
            "readOnly": false,
            "resources": [
                {
                    "accountId": "111122223333",
                    "type": "AWS::DynamoDB::Table",
                    "ARN": "arn:aws:dynamodb:us-west-2:123456789012:table/Music"
                }
            ],
```

```
            "eventType": "AwsApiCall",
            "apiVersion": "2012-08-10",
            "managementEvent": false,
            "recipientAccountId": "111122223333",
            "eventCategory": "Data"
        }
    ]
}
```

`UpdateItem` (Unsuccessful)

```
{
    "Records": [
        {
            "eventVersion": "1.07",
            "userIdentity": {
                "type": "AssumedRole",
                "principalId": "AKIAIOSFODNN7EXAMPLE:bob",
                "arn": "arn:aws:sts::111122223333:assumed-role/users/bob",
                "accountId": "111122223333",
                "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
                "sessionContext": {
                    "sessionIssuer": {
                        "type": "Role",
                        "principalId": "AKIAI44QH8DHBEXAMPLE",
                        "arn": "arn:aws:iam::444455556666:role/admin-role",
                        "accountId": "444455556666",
                        "userName": "bob"
                    },
                    "attributes": {
                        "creationDate": "2020-09-03T22:14:13Z",
                        "mfaAuthenticated": "false"
                    }
                }
            },
            "eventTime": "2020-09-03T22:27:15Z",
            "eventSource": "dynamodb.amazonaws.com",
            "eventName": "UpdateItem",
            "awsRegion": "us-west-2",
            "sourceIPAddress": "192.0.2.0",
            "userAgent": "aws-cli/1.15.64 Python/2.7.16 Darwin/17.7.0
 botocore/1.10.63",
            "errorCode": "ConditionalCheckFailedException",
```

```
            "errorMessage": "The conditional request failed",
            "requestParameters": {
                "tableName": "Music",
                "key": {
                    "Artist": "No One You Know",
                    "SongTitle": "Call Me Today"
                },
                "updateExpression": "SET #Y = :y, #AT = :t",
                "expressionAttributeNames": {
                    "#Y": "Year",
                    "#AT": "AlbumTitle"
                },
                "conditionExpression": "attribute_not_exists(#Y)",
                "returnConsumedCapacity": "TOTAL"
            },
            "responseElements": null,
            "requestID": "4KBNVRGD25RG1KEO9UT4V3FQDJVV4KQNSO5AEMVJF66Q9ASUAAJG",
            "eventID": "a954451c-c2fc-4561-8aea-7a30ba1fdf52",
            "readOnly": false,
            "resources": [
                {
                    "accountId": "111122223333",
                    "type": "AWS::DynamoDB::Table",
                    "ARN": "arn:aws:dynamodb:us-west-2:123456789012:table/Music"
                }
            ],
            "eventType": "AwsApiCall",
            "apiVersion": "2012-08-10",
            "managementEvent": false,
            "recipientAccountId": "111122223333",
            "eventCategory": "Data"
        }
    ]
}
```


`TransactWriteItems` (Successful)


```
{
    "Records": [
        {
            "eventVersion": "1.07",
            "userIdentity": {
                "type": "AssumedRole",
```

```
                "principalId": "AKIAIOSFODNN7EXAMPLE:bob",
                "arn": "arn:aws:sts::111122223333:assumed-role/users/bob",
                "accountId": "111122223333",
                "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
                "sessionContext": {
                    "sessionIssuer": {
                        "type": "Role",
                        "principalId": "AKIAI44QH8DHBEXAMPLE",
                        "arn": "arn:aws:iam::444455556666:role/admin-role",
                        "accountId": "444455556666",
                        "userName": "bob"
                    },
                    "attributes": {
                        "creationDate": "2020-09-03T22:14:13Z",
                        "mfaAuthenticated": "false"
                    }
                }
            },
            "eventTime": "2020-09-03T21:48:12Z",
            "eventSource": "dynamodb.amazonaws.com",
            "eventName": "TransactWriteItems",
            "awsRegion": "us-west-1",
            "sourceIPAddress": "192.0.2.0",
            "userAgent": "aws-cli/1.15.64 Python/2.7.16 Darwin/17.7.0
botocore/1.10.63",
            "requestParameters": {
                "requestItems": [
                    {
                        "operation": "Put",
                        "tableName": "Music",
                        "key": {
                            "Artist": "No One You Know",
                            "SongTitle": "Call Me Today"
                        },
                        "items": [
                            "Artist",
                            "SongTitle",
                            "AlbumTitle"
                        ],
                        "conditionExpression": "#AT = :A",
                        "expressionAttributeNames": {
                            "#AT": "AlbumTitle"
                        },
                        "returnValuesOnConditionCheckFailure": "ALL_OLD"
```

```
            },
            {
                "operation": "Update",
                "tableName": "Music",
                "key": {
                    "Artist": "No One You Know",
                    "SongTitle": "Call Me Tomorrow"
                },
                "updateExpression": "SET #AT = :newval",
                "ConditionExpression": "attribute_not_exists(Rating)",
                "ExpressionAttributeNames": {
                    "#AT": "AlbumTitle"
                },
                "returnValuesOnConditionCheckFailure": "ALL_OLD"
            },
            {
                "operation": "Delete",
                "TableName": "Music",
                "key": {
                    "Artist": "No One You Know",
                    "SongTitle": "Call Me Yesterday"
                },
                "conditionExpression": "#P between :lo and :hi",
                "expressionAttributeNames": {
                    "#P": "Price"
                },
                "ReturnValuesOnConditionCheckFailure": "ALL_OLD"
            },
            {
                "operation": "ConditionCheck",
                "TableName": "Music",
                "Key": {
                    "Artist": "No One You Know",
                    "SongTitle": "Call Me Now"
                },
                "ConditionExpression": "#P between :lo and :hi",
                "ExpressionAttributeNames": {
                    "#P": "Price"
                },
                "ReturnValuesOnConditionCheckFailure": "ALL_OLD"
            }
        ],
        "returnConsumedCapacity": "TOTAL",
        "returnItemCollectionMetrics": "SIZE"
```

```
            },
            "responseElements": null,
            "requestID": "45EN32OM6TQSMV2MI65O4L5TNFVV4KQNSO5AEMVJF66Q9ASUAAJG",
            "eventID": "4f1cc78b-5c94-4174-a6ad-3ee78605381c",
            "readOnly": false,
            "resources": [
                {
                    "accountId": "111122223333",
                    "type": "AWS::DynamoDB::Table",
                    "ARN": "arn:aws:dynamodb:us-west-2:123456789012:table/Music"
                }
            ],
            "eventType": "AwsApiCall",
            "apiVersion": "2012-08-10",
            "managementEvent": false,
            "recipientAccountId": "111122223333",
            "eventCategory": "Data"
        }
    ]
}
```

`TransactWriteItems` (With `TransactionCanceledException`)

```
{
    "Records": [
        {
            "eventVersion": "1.06",
            "userIdentity": {
                "type": "AssumedRole",
                "principalId": "AKIAIOSFODNN7EXAMPLE:bob",
                "arn": "arn:aws:sts::111122223333:assumed-role/users/bob",
                "accountId": "111122223333",
                "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
                "sessionContext": {
                    "sessionIssuer": {
                        "type": "Role",
                        "principalId": "AKIAI44QH8DHBEXAMPLE",
                        "arn": "arn:aws:iam::444455556666:role/admin-role",
                        "accountId": "444455556666",
                        "userName": "bob"
                    },
                    "attributes": {
                        "creationDate": "2020-09-03T22:14:13Z",
```

```
                    "mfaAuthenticated": "false"
                }
            }
        },
        "eventTime": "2019-02-01T00:42:34Z",
        "eventSource": "dynamodb.amazonaws.com",
        "eventName": "TransactWriteItems",
        "awsRegion": "us-west-2",
        "sourceIPAddress": "192.0.2.0",
        "userAgent": "aws-cli/1.16.93 Python/3.4.7
Linux/4.9.119-0.1.ac.277.71.329.metal1.x86_64 botocore/1.12.83",
        "errorCode": "TransactionCanceledException",
        "errorMessage": "Transaction cancelled, please refer cancellation reasons
for specific reasons [ConditionalCheckFailed, None]",
        "requestParameters": {
            "requestItems": [
                {
                    "operation": "Put",
                    "tableName": "Music",
                    "key": {
                        "Artist": "No One You Know",
                        "SongTitle": "Call Me Today"
                    },
                    "items": [
                        "Artist",
                        "SongTitle",
                        "AlbumTitle"
                    ],
                    "conditionExpression": "#AT = :A",
                    "expressionAttributeNames": {
                        "#AT": "AlbumTitle"
                    },
                    "returnValuesOnConditionCheckFailure": "ALL_OLD"
                },
                {
                    "operation": "Update",
                    "tableName": "Music",
                    "key": {
                        "Artist": "No One You Know",
                        "SongTitle": "Call Me Tomorrow"
                    },
                    "updateExpression": "SET #AT = :newval",
                    "ConditionExpression": "attribute_not_exists(Rating)",
                    "ExpressionAttributeNames": {
```

```
                        "#AT": "AlbumTitle"
                    },
                    "returnValuesOnConditionCheckFailure": "ALL_OLD"
                },
                {
                    "operation": "Delete",
                    "TableName": "Music",
                    "key": {
                        "Artist": "No One You Know",
                        "SongTitle": "Call Me Yesterday"
                    },
                    "conditionExpression": "#P between :lo and :hi",
                    "expressionAttributeNames": {
                        "#P": "Price"
                    },
                    "ReturnValuesOnConditionCheckFailure": "ALL_OLD"
                },
                {
                    "operation": "ConditionCheck",
                    "TableName": "Music",
                    "Key": {
                        "Artist": "No One You Know",
                        "SongTitle": "Call Me Now"
                    },
                    "ConditionExpression": "#P between :lo and :hi",
                    "ExpressionAttributeNames": {
                        "#P": "Price"
                    },
                    "ReturnValuesOnConditionCheckFailure": "ALL_OLD"
                }
            ],
            "returnConsumedCapacity": "TOTAL",
            "returnItemCollectionMetrics": "SIZE"
        },
        "responseElements": null,
        "requestID": "A0GTQEKLBB9VD8E05REA5A3E1VVV4KQNSO5AEMVJF66Q9ASUAAJG",
        "eventID": "43e437b5-908a-46af-84e6-e27fffb9c5cd",
        "readOnly": false,
        "resources": [
            {
                "accountId": "111122223333",
                "type": "AWS::DynamoDB::Table",
                "ARN": "arn:aws:dynamodb:us-west-2:123456789012:table/Music"
            }
```

```
            ],
            "eventType": "AwsApiCall",
            "apiVersion": "2012-08-10",
            "managementEvent": false,
            "recipientAccountId": "111122223333",
            "eventCategory": "Data"
        }
    ]
}
```

ExecuteStatement

```
{
    "Records": [
        {
            "eventVersion": "1.08",
            "userIdentity": {
                "type": "AssumedRole",
                "principalId": "AKIAIOSFODNN7EXAMPLE:bob",
                "arn": "arn:aws:sts::111122223333:assumed-role/users/bob",
                "accountId": "111122223333",
                "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
                "sessionContext": {
                    "sessionIssuer": {
                        "type": "Role",
                        "principalId": "AKIAI44QH8DHBEXAMPLE",
                        "arn": "arn:aws:iam::444455556666:role/admin-role",
                        "accountId": "444455556666",
                        "userName": "bob"
                    },
                    "attributes": {
                        "creationDate": "2020-09-03T22:14:13Z",
                        "mfaAuthenticated": "false"
                    }
                }
            },
            "eventTime": "2021-03-03T23:06:45Z",
            "eventSource": "dynamodb.amazonaws.com",
            "eventName": "ExecuteStatement",
            "awsRegion": "us-west-2",
            "sourceIPAddress": "192.0.2.0",
            "userAgent": "aws-cli/1.19.7 Python/3.6.13
 Linux/4.9.230-0.1.ac.223.84.332.metal1.x86_64 botocore/1.20.7",
```

```
            "requestParameters": {
                "statement": "SELECT * FROM Music WHERE Artist = 'No One You Know' AND
 SongTitle = 'Call Me Today' AND nonKeyAttr = ***(Redacted)"
            },
            "responseElements": null,
            "requestID": "V7G2KCSFLP83ORB7MMFG6RIAD3VV4KQNSO5AEMVJF66Q9ASUAAJG",
            "eventID": "0b5c4779-e169-4227-a1de-6ed01dd18ac7",
            "readOnly": false,
            "resources": [
                {
                    "accountId": "111122223333",
                    "type": "AWS::DynamoDB::Table",
                    "ARN": "arn:aws:dynamodb:us-west-2:123456789012:table/Music"
                }
            ],
            "eventType": "AwsApiCall",
            "apiVersion": "2012-08-10",
            "managementEvent": false,
            "recipientAccountId": "111122223333",
            "eventCategory": "Data"
        }
    ]
}
```

BatchExecuteStatement

```
{
    "Records": [
        {
            "eventVersion": "1.08",
            "userIdentity": {
                "type": "AssumedRole",
                "principalId": "AKIAIOSFODNN7EXAMPLE:bob",
                "arn": "arn:aws:sts::111122223333:assumed-role/users/bob",
                "accountId": "111122223333",
                "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
                "sessionContext": {
                    "sessionIssuer": {
                        "type": "Role",
                        "principalId": "AKIAI44QH8DHBEXAMPLE",
                        "arn": "arn:aws:iam::444455556666:role/admin-role",
                        "accountId": "444455556666",
                        "userName": "bob"
```

```
                    },
                    "attributes": {
                        "creationDate": "2020-09-03T22:14:13Z",
                        "mfaAuthenticated": "false"
                    }
                }
            },
            "eventTime": "2021-03-03T23:24:48Z",
            "eventSource": "dynamodb.amazonaws.com",
            "eventName": "BatchExecuteStatement",
            "awsRegion": "us-west-2",
            "sourceIPAddress": "192.0.2.0",
            "userAgent": "aws-cli/1.19.7 Python/3.6.13
Linux/4.9.230-0.1.ac.223.84.332.metal1.x86_64 botocore/1.20.7",
            "requestParameters": {
                "requestItems": [
                    {
                        "statement": "UPDATE Music SET Album = ***(Redacted) WHERE
Artist = 'No One You Know' AND SongTitle = 'Call Me Today'"
                    },
                    {
                        "statement": "INSERT INTO Music VALUE {'Artist' :
***(Redacted), 'SongTitle' : ***(Redacted), 'Album' : ***(Redacted)}"
                    }
                ]
            },
            "responseElements": null,
            "requestID": "23PE7ED291UD65P9SMS6TISNVBVV4KQNSO5AEMVJF66Q9ASUAAJG",
            "eventID": "f863f966-b741-4c36-b15e-f867e829035a",
            "readOnly": false,
            "resources": [
                {
                    "accountId": "111122223333",
                    "type": "AWS::DynamoDB::Table",
                    "ARN": "arn:aws:dynamodb:us-west-2:123456789012:table/Music"
                }
            ],
            "eventType": "AwsApiCall",
            "apiVersion": "2012-08-10",
            "managementEvent": false,
            "recipientAccountId": "111122223333",
            "eventCategory": "Data"
        }
    ]
```

```
}
```

GetRecords

```
{
    "Records": [
        {
            "eventVersion": "1.08",
            "userIdentity": {
                "type": "AssumedRole",
                "principalId": "AKIAIOSFODNN7EXAMPLE:bob",
                "arn": "arn:aws:sts::111122223333:assumed-role/users/bob",
                "accountId": "111122223333",
                "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
                "sessionContext": {
                    "sessionIssuer": {
                        "type": "Role",
                        "principalId": "AKIAI44QH8DHBEXAMPLE",
                        "arn": "arn:aws:iam::444455556666:role/admin-role",
                        "accountId": "444455556666",
                        "userName": "bob"
                    },
                    "attributes": {
                        "creationDate": "2020-09-03T22:14:13Z",
                        "mfaAuthenticated": "false"
                    }
                }
            },
            "eventTime": "2021-04-15T04:15:02Z",
            "eventSource": "dynamodb.amazonaws.com",
            "eventName": "GetRecords",
            "awsRegion": "us-west-2",
            "sourceIPAddress": "192.0.2.0",
            "userAgent": "aws-cli/1.19.50 Python/3.6.13
 Linux/4.9.230-0.1.ac.224.84.332.metal1.x86_64 botocore/1.20.50",
            "requestParameters": {
                "shardIterator": "arn:aws:dynamodb:us-west-2:123456789012:table/
Music/stream/2021-04-15T04:02:47.428|1|AAAAAAAAAAH7HF3xwDQHBrvk2UBZ1PKh8bX3F
+JeH0rFwHCE7dz4VGV1ZoJ5bMxQwkmerA3wzCTL+zSseGLdSXNJP14EwrjLNvDNoZeRSJ/
n6xc3I4NYOptR4zR8d7VrjMAD6h5nR12NtxGIgJ/
dVsUpluWsHyCW3PPbKsMlJSruVRWoitRhSd3S6slEWEPB0bDC7+
+ISH5mXrCHOnvyezQKlqNshTSPZ5jWwqRj2VNSXCMTGXv9P01/
U0bpOUI2cuRTchgUpPSe3ur2sQrRj3KlbmIyCz7P
```

```
+H3CYlugafi8fQ5kipDSkESkIWS6O5ejzibWKg/3izms1eVIm/
zLFdEeihCYJ7G8fpHUSLX5JAk3ab68aUXGSFEZLONntgNIhQkcMo00/
mJlaIgkEdBUyqvZO1vtKUBH5YonIrZqSUhv8Coc+mh24vOg1YI+SPIXlr
+Lnl54BG6AjrmaScjHACVXoPDxPsXSJXC4c9HjoC3YSskCPV7uWi0f65/
n7JAT3cskcX2ISaLHwYzJPaMBSftxOgeRLm3BnisL32nT8uTj2gF/
PUrEjdyoqTX7EerQpcaekXmOgay5Kh8n4T2uPdM83f356vRpar/
DDp8pLFD0ddb6Yvz7zU2zGdAvTod3IScC1GpTqcjRxaMhlBVZy1TnI9Cs
+7fXMdUF6xYScjR2725icFBNLojSFVDmsfHabXaCEpmeuXZsLbp5CjcPAHa66R8mQ5tSoFjrzOEzeB4uconEXAMPLE=="
            },
            "responseElements": null,
            "requestID": "1M0U1Q80P4LDPT7A7N1A758N2VVV4KQNSO5AEMVJF66Q9EXAMPLE",
            "eventID": "09a634f2-da7d-4c9e-a259-54aceexample",
            "readOnly": true,
            "resources": [
                {
                    "accountId": "111122223333",
                    "type": "AWS::DynamoDB::Table",
                    "ARN": "arn:aws:dynamodb:us-west-2:123456789012:table/Music"
                }
            ],
            "eventType": "AwsApiCall",
            "apiVersion": "2012-08-10",
            "managementEvent": false,
            "recipientAccountId": "111122223333",
            "eventCategory": "Data"
        }
    ]
}
```


Analyzing data access using CloudWatch contributor insights for DynamoDB

Amazon CloudWatch Contributor Insights for Amazon DynamoDB is a diagnostic tool for identifying the most frequently accessed and throttled keys in your table or index at a glance. This tool uses CloudWatch contributor insights.

By enabling CloudWatch Contributor Insights for DynamoDB on a table or global secondary index, you can view the most accessed and throttled items in those resources.

> **Note**
>
> CloudWatch charges apply for Contributor Insights for DynamoDB. For more information about pricing, see Amazon CloudWatch pricing.

Topics

- CloudWatch contributor insights for DynamoDB: How it works
- Getting started with CloudWatch Contributor Insights for DynamoDB
- Using IAM with CloudWatch contributor insights for DynamoDB

CloudWatch contributor insights for DynamoDB: How it works

Amazon DynamoDB integrates with CloudWatch Contributor Insights to provide information about the most accessed and throttled items in a table or global secondary index. DynamoDB delivers this information to you via CloudWatch Contributor Insights rules, reports, and graphs of report data.

CloudWatch Contributor Insights for DynamoDB is designed to have no performance impact on your DynamoDB table.

For more information about CloudWatch Contributor Insights, see Using Contributor Insights to analyze high-cardinality data in the *Amazon CloudWatch User Guide*.

The following sections describe the core concepts and behavior of CloudWatch Contributor Insights for DynamoDB.

Topics

- CloudWatch contributor insights modes for DynamoDB
- CloudWatch contributor insights for DynamoDB rules
- Understanding CloudWatch contributor insights for DynamoDB graphs
- Interactions with other DynamoDB features
- CloudWatch contributor insights for DynamoDB billing

CloudWatch contributor insights modes for DynamoDB

CloudWatch Contributor Insights for DynamoDB offers two distinct modes to meet different monitoring needs.

Throttled keys mode

This mode focuses exclusively on throttled requests by only processing events when throttling occurs. It delivers insights about performance issues without the overhead of tracking all access patterns. In this mode, DynamoDB tracks only the:

- **Most throttled items** — Items that experience the most throttling events

This mode is ideal when:

- Your primary concern is identifying and resolving throttling problems
- You want to keep Contributor Insights enabled continuously for real-time throttling detection
- You want a cost-optimized approach to monitor throttling issues

> **Note**
>
> *Throttled keys mode* processes events only when throttling occurs, making it cost-effective for continuous monitoring. This targeted approach allows you to leave the feature enabled permanently with minimal cost impact, while still providing immediate visibility into throttling issues as they happen.

If your table experiences no throttling, you won't see any data in the Contributor Insights graphs, which indicates healthy performance. When throttling is detected, the generated graphs help you identify specific access patterns causing performance issues. This information can help you implement strategies to address non-uniform access patterns.

For comprehensive monitoring strategies, you can integrate these throttling insights with other CloudWatch metrics to create unified dashboards that correlate throttling events with overall table performance.

Accessed and throttled keys mode

This mode provides comprehensive monitoring of both accessed and throttled items. In this mode, DynamoDB tracks the:

- **Most accessed items** — Items that consume the most read and write capacity
- **Most throttled items** — Items that experience the most throttling events

This mode is ideal when you need complete visibility into your table's access patterns and want to understand both high-traffic items and throttling issues.

Switching between modes

You can switch between modes at any time using the DynamoDB console, AWS CLI, or APIs. When you switch modes:

- Existing CloudWatch rules are updated to match the new mode
- Throttled keys CloudWatch rules remain intact, maintaining your continuous historical data for throttling metrics:
  - When you switch from *throttled keys* mode to *accessed and throttled keys* mode, the existing throttled key rules are preserved, and new accessed key rules are created
  - When you switch from *accessed and throttled keys* mode to *throttled keys* mode, only the throttled key rules are preserved, and the accessed key rules are removed
- Billing adjusts immediately to reflect the new mode's event processing

CloudWatch contributor insights for DynamoDB rules

When you enable CloudWatch Contributor Insights for DynamoDB on a table or global secondary index, DynamoDB creates rules on your behalf based on the selected mode.

> **Note**
>
> When you enable Contributor Insights on your DynamoDB table, you're subject to Contributor Insights rules limits. For more information, see CloudWatch service quotas.

Rules for accessed and throttled keys mode

In *accessed and throttled keys* mode, DynamoDB creates the following rules:

- **Most accessed items (partition keys)** — Identifies the partition keys of the most accessed items in your table or global secondary index.

  CloudWatch rule name format: `DynamoDBContributorInsights-PKC-[resource_name]-[creationtimestamp]`

- **Most throttled keys (partition keys)** — Identifies the partition keys of the most throttled items in your table or global secondary index.

  CloudWatch rule name format: `DynamoDBContributorInsights-PKT-[resource_name]-[creationtimestamp]`

If your table or global secondary index has sort keys, DynamoDB also creates the following rules specific to sort keys:

- **Most accessed keys (partition and sort keys)** — Identifies the partition and sort keys of the most accessed items in your table or global secondary index.

  CloudWatch rule name format: `DynamoDBContributorInsights-SKC-[resource_name]-[creationtimestamp]`
- **Most throttled keys (partition and sort keys)** — Identifies the partition and sort keys of the most throttled items in your table or global secondary index.

  CloudWatch rule name format: `DynamoDBContributorInsights-SKT-[resource_name]-[creationtimestamp]`

Rules for throttled keys mode

In *throttled keys* mode, DynamoDB creates only the throttling-related rules:

- **Most throttled keys (partition key)** — Identifies the partition keys of the most throttled items in your table or global secondary index.

  CloudWatch rule name format: `DynamoDBContributorInsights-PKT-[resource_name]-[creationtimestamp]`

If your table or global secondary index has sort keys, DynamoDB also creates:

- **Most throttled keys (partition and sort keys)** — Identifies the partition and sort keys of the most throttled items in your table or global secondary index.

  CloudWatch rule name format: `DynamoDBContributorInsights-SKT-[resource_name]-[creationtimestamp]`

This focused approach reduces the number of active rules and decreases the volume of events processed to better diagnose your throttling events.

Note

- When you use the CloudWatch console or APIs to view CloudWatch Contributor Insights for DynamoDB, you only see rules corresponding to your selected mode.
- You can't use the CloudWatch console or APIs to directly modify or delete the rules created by CloudWatch Contributor Insights for DynamoDB. Disabling CloudWatch Contributor Insights for DynamoDB on a table or global secondary index automatically deletes the rules created for that table or global secondary index.
- When you use the GetInsightRuleReport operation with CloudWatch Contributor Insights rules that are created by DynamoDB, only `MaxContributorValue` and `Maximum` return useful statistics. The other statistics in this list don't return meaningful values.
- CloudWatch Contributor Insights for DynamoDB has a limit of 25 contributors. Requesting more than 25 contributors will return an error.

You can create CloudWatch Alarms using the CloudWatch Contributor Insights for DynamoDB rules. This allows you to be notified when any item exceed or meets a specific threshold for `ConsumedThroughputUnits` or `ThrottleCount`. For more information, see Setting an alarm on Contributor Insights metric data.

Understanding CloudWatch contributor insights for DynamoDB graphs

CloudWatch Contributor Insights for DynamoDB displays different types of graphs on both the DynamoDB and CloudWatch consoles depending on the selected mode.

Graph availability by mode

The graphs displayed depend on your selected Contributor Insights mode.

- **Accessed and throttled keys mode** displays both *Most Accessed Items* and *Most Throttled Items* graphs
- **Throttled keys mode** displays only *Most Throttled Items* graphs

Most accessed items

This graph is available only in accessed and throttled keys mode. Use this graph to identify the most accessed items in the table or global secondary index. The graph displays `ConsumedThroughputUnits` on the y-axis and time on the x-axis. Each of the top N keys is displayed in its own color, with a legend displayed below the x-axis.

DynamoDB measures key access frequency by using `ConsumedThroughputUnits`, which measures combined read and write traffic. `ConsumedThroughputUnits` is defined as the following:

- Provisioned — *(3 x consumed write capacity units) + consumed read capacity units*
- On-demand — *(3 x write request units) + read request units*

On the DynamoDB console, each data point in the graph represents the maximum of `ConsumedThroughputUnits` over a 1-minute period. For example, a graph value of 180,000 `ConsumedThroughputUnits` indicates that the item was accessed continuously at the per-item maximum throughput of 1,000 write request units or 3,000 read request units for a 60-second span within that 1-minute period (3,000 x 60 seconds). In other words, the graphed values represent the highest-traffic minute within each 1-minute period. You can change the time granularity of the `ConsumedThroughputUnits` metric (for example, to view 5-minute metrics instead of 1-minute) on the CloudWatch console.

If you see several closely clustered lines without any obvious outliers, it indicates that your workload is relatively balanced across items over the given time window. If you see isolated points in the graph instead of connected lines, it indicates an item that was frequently accessed only for a brief period.

If your table or global secondary index has sort keys, DynamoDB creates two graphs: one for the most accessed partition keys and one for the most accessed partition + sort keys pairs. You can see traffic at the partition keys level in the partition key–only graph. You can see traffic at the item level in the partition + sort keys graphs.

Most throttled items

This graph is available in both modes. Use this graph to identify the most throttled items in the table or global secondary index. The graph displays `ThrottleCount` on the y-axis and time on the x-axis. Each of the top *N* keys is displayed in its own color, with a legend displayed below the x-axis.

DynamoDB measures throttle frequency using `ThrottleCount`, which is the count of `ProvisionedThroughputExceededException`, `ThrottlingException`, and `RequestLimitExceeded` errors.

Write throttling caused by insufficient write capacity for a global secondary index is not measured. You can use the *Most Accessed Items* graph of the global secondary index to identify imbalanced access patterns that may cause write throttling. For more information, see Provisioned throughput Considerations for Global Secondary Indexes.

On the DynamoDB console, each data point in the graph represents the count of throttle events over a 1-minute period.

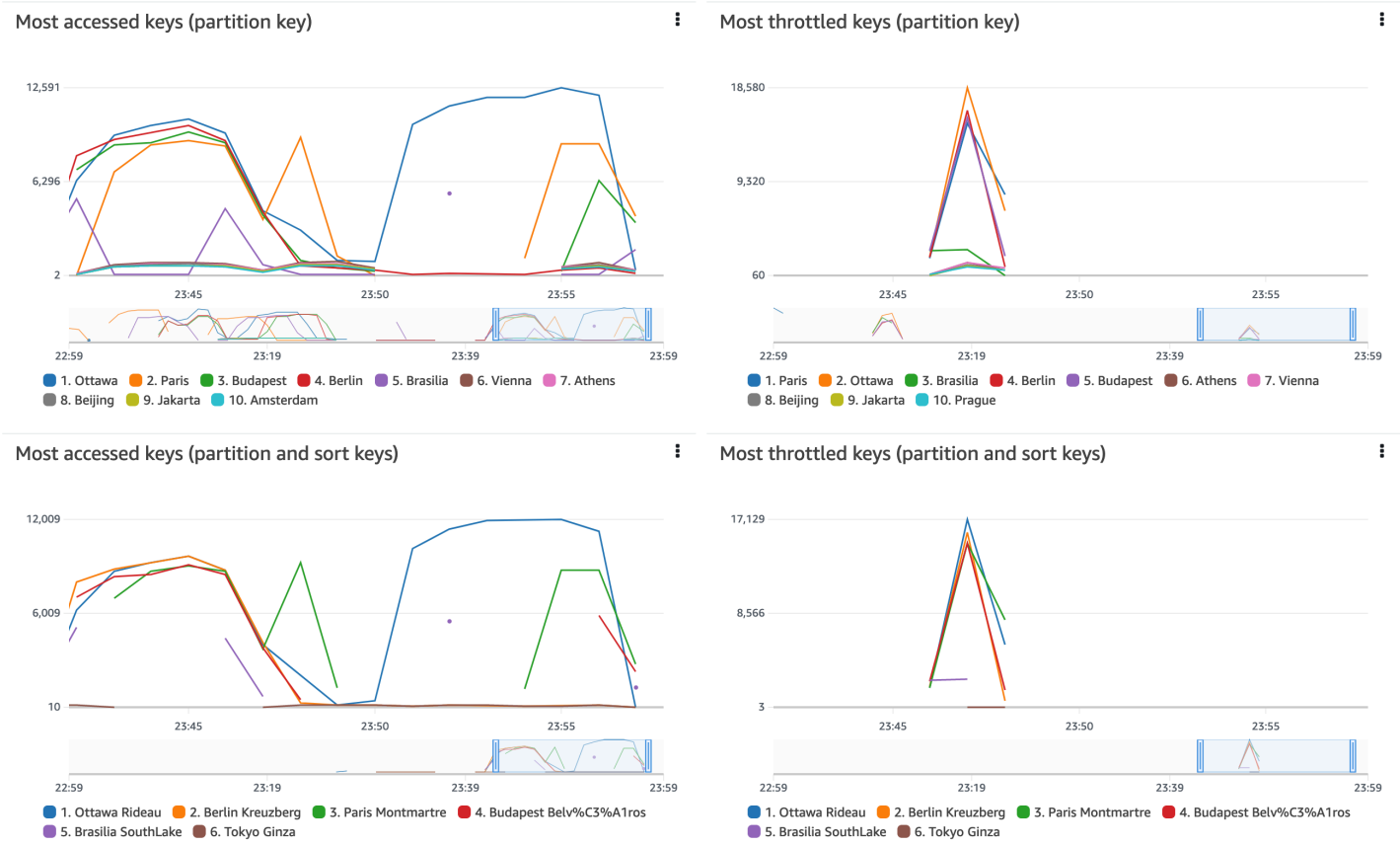
If you see no data in this graph, it indicates that your requests are not being throttled. If you see isolated points in the graph instead of connected lines, it indicates that an item was frequently throttled for a brief period.

If your table or global secondary index has sort keys, DynamoDB creates two graphs: one for most throttled partition keys and one for most throttled partition + sort keys pairs. You can see throttle count at the partition keys level in the partition keys-only graph, and throttle count at the item-level in the partition + sort keys graphs.

> **Note**
>
> In *throttled keys* mode, this is the only type of graph you'll see. The absence of data in these graphs indicates healthy table performance with no throttling occurring.

Report examples

The following example shows the reports generated for a table with both a partition keys and sort keys in *accessed and throttled keys* modes. In *throttled keys* mode, you see only the throttling-related portion of this report.



Interactions with other DynamoDB features

The following sections describe how CloudWatch Contributor Insights for DynamoDB behaves and interacts with several other features in DynamoDB. These behaviors apply to both modes unless otherwise specified.

Global tables

CloudWatch Contributor Insights for DynamoDB monitors global table replicas as distinct tables. The Contributor Insights graphs for a replica in one AWS Region might not show the same patterns as another Region. This is because write data is replicated across all replicas in a global table, but each replica can serve Region-bound read traffic.

Each replica can be configured with a different Contributor Insights mode independently. For example, you might use *accessed and throttled keys* mode in your primary region for comprehensive monitoring, while using *throttled keys* mode in secondary regions to maintain visibility into performance issues.

DynamoDB Accelerator (DAX)

CloudWatch Contributor Insights for DynamoDB doesn't show DAX cache responses. It only shows responses to accessing a table or a global secondary index.

> **Note**
>
> DynamoDB CloudWatch Contributor Insights does not support PartiQL requests.

Encryption at rest

CloudWatch Contributor Insights for DynamoDB doesn't affect how encryption works in DynamoDB. The primary key data that is published in CloudWatch is encrypted with the AWS owned key. However, DynamoDB also supports the AWS managed key and a customer managed key.

CloudWatch Contributor Insights for DynamoDB displays partition keys and sort keys (if applicable) of frequently accessed and throttled items. While CloudWatch Contributor Insights works with encrypted DynamoDB tables, it's important to note that it uses its own Amazon-owned encryption context, which is separate from the table's configured encryption.

If your DynamoDB table's primary key contains sensitive information and your organization's security policies require full control over encryption processes, enabling CloudWatch Contributor Insights may not be suitable.

Fine-grained access control

CloudWatch Contributor Insights for DynamoDB doesn't function differently for tables with fine-grained access control (FGAC). In other words, any user who has the appropriate CloudWatch permissions can view FGAC-protected primary keys in CloudWatch Contributor Insights graphs.

If the table's primary key contains FGAC-protected data that you don't want published to CloudWatch, you should not enable CloudWatch Contributor Insights for DynamoDB for that table.

Access control

You control access to CloudWatch Contributor Insights for DynamoDB using AWS Identity and Access Management (IAM) by limiting DynamoDB control plane permissions and CloudWatch data plane permissions. For more information see, Using IAM with CloudWatch Contributor Insights for DynamoDB.

CloudWatch contributor insights for DynamoDB billing

Charges for CloudWatch Contributor Insights for DynamoDB appear in the CloudWatch section of your monthly bill. These charges are calculated based on the number of DynamoDB events that are processed, and the selected mode.

Billing by mode

The two Contributor Insights modes have different billing characteristics.

- **Accessed and throttled keys mode billing** - In this mode, each item that is written or read via a data plane operation represents one event, regardless of whether the request succeeds or is throttled. If a table or global secondary index includes sort keys, each item that is read or written represents two events. This is because DynamoDB is identifying top contributors from separate time series: one for partitions keys only, and one for partition and sort keys pairs.
- **Throttled keys mode billing** - In this mode, only throttled requests generate billable events. Events are only generated when requests result in `ProvisionedThroughputExceededException`, `ThrottlingException`, or `RequestLimitExceeded` errors. If a table or global secondary index includes sort keys, each throttled item represents two events (partition keys tracking and partition + sort keys tracking).

Billing examples

For example, assume that your application performs the following DynamoDB operations: a `GetItem`, a `PutItem`, and a `BatchWriteItem` that puts five items. Also assume that the `PutItem` operation gets throttled, but all other operations succeed.

- **Accessed and throttled keys mode**
  - If your table or global secondary index has only a partition keys, it results in 7 events (1 for the `GetItem`, 1 for the `PutItem`, and 5 for the `BatchWriteItem`).
  - If your table or global secondary index has a partition keys and sort keys, it results in 14 events (2 for the `GetItem`, 2 for the `PutItem`, and 10 for the `BatchWriteItem`).
- **Throttled keys mode**
  - If your table or global secondary index has only a partition keys, it results in 1 event (only for the throttled `PutItem`).
  - If your table or global secondary index has a partition keys and sort keys, it results in 2 events (2 for the throttled `PutItem`).

The successful `GetItem` and `BatchWriteItem` operations generate no events in throttled keys mode.

Common billing factors

A `Query` operation always results in 1 event, regardless of the mode or number of items returned.

Unlike other DynamoDB features, CloudWatch Contributor Insights for DynamoDB billing *does not* vary based on the following:

- The capacity mode (provisioned vs. on-demand)
- Whether you perform read or write requests
- The size (KB) of the items read or written

Getting started with CloudWatch Contributor Insights for DynamoDB

This section describes how to enable and use Amazon CloudWatch Contributor Insights in different modes to meet your monitoring needs using the Amazon DynamoDB console or the AWS Command Line Interface (AWS CLI).

In the following examples, you use the DynamoDB table that is defined in the Getting started with DynamoDB tutorial.

Topics

- Choosing a Contributor Insights mode
- Using Contributor Insights (console)
- Using Contributor Insights (AWS CLI)

Choosing a Contributor Insights mode

Before enabling Contributor Insights, you should understand the two available modes. Review the mode comparison to select the option that best aligns with your specific requirements.

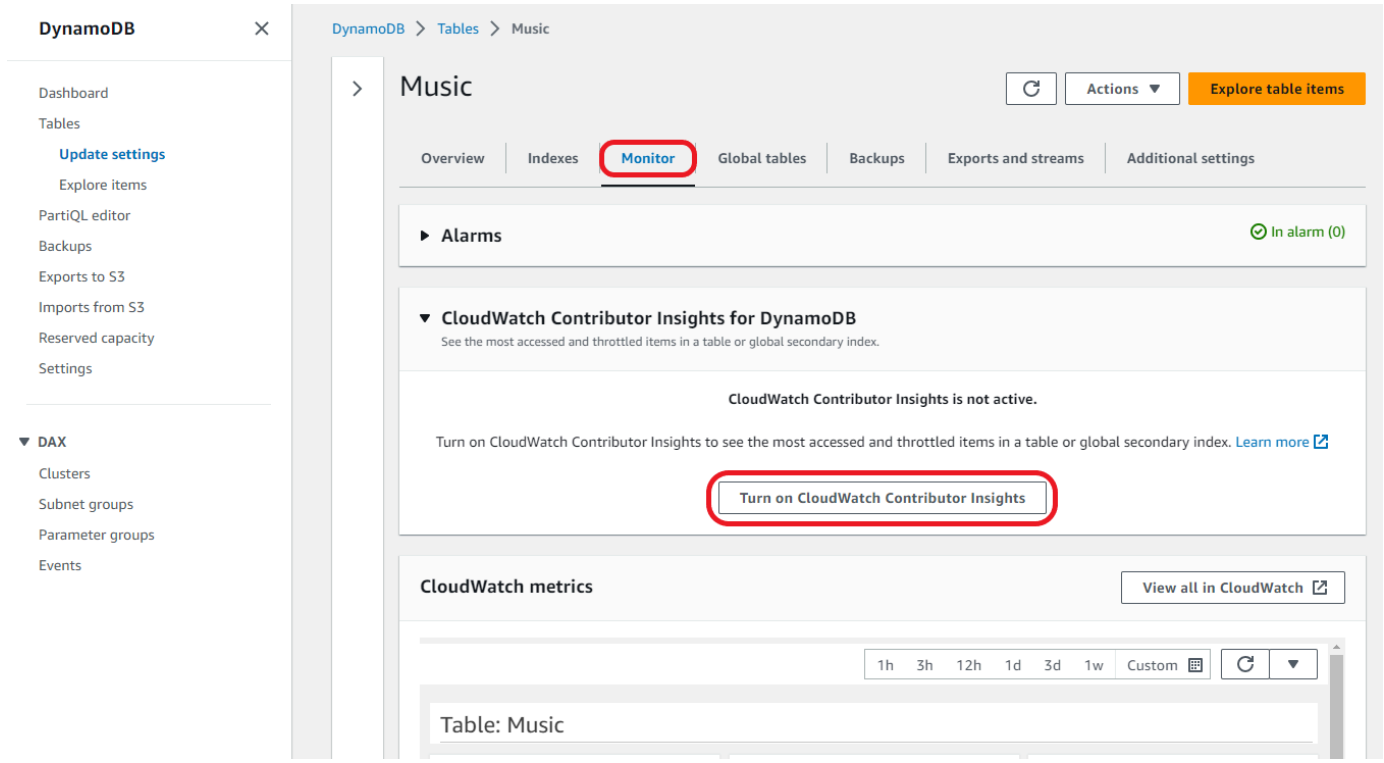
Aspect	Accessed and throttled keys mode	Throttled keys mode
Monitors	All requests (successful and throttled)	Only throttled requests
Graphs	Most Accessed Items + Most Throttled Items	Most Throttled Items only
Best for	Targeted analysis and optimization	Throttling monitoring
Use when	You need complete visibility into access patterns. You're doing short-term analysis or debugging.	Your primary concern is identifying and resolving throttling issues. You want to keep Contributor Insights enabled continuously for real-time throttling alerts.

Using Contributor Insights (console)

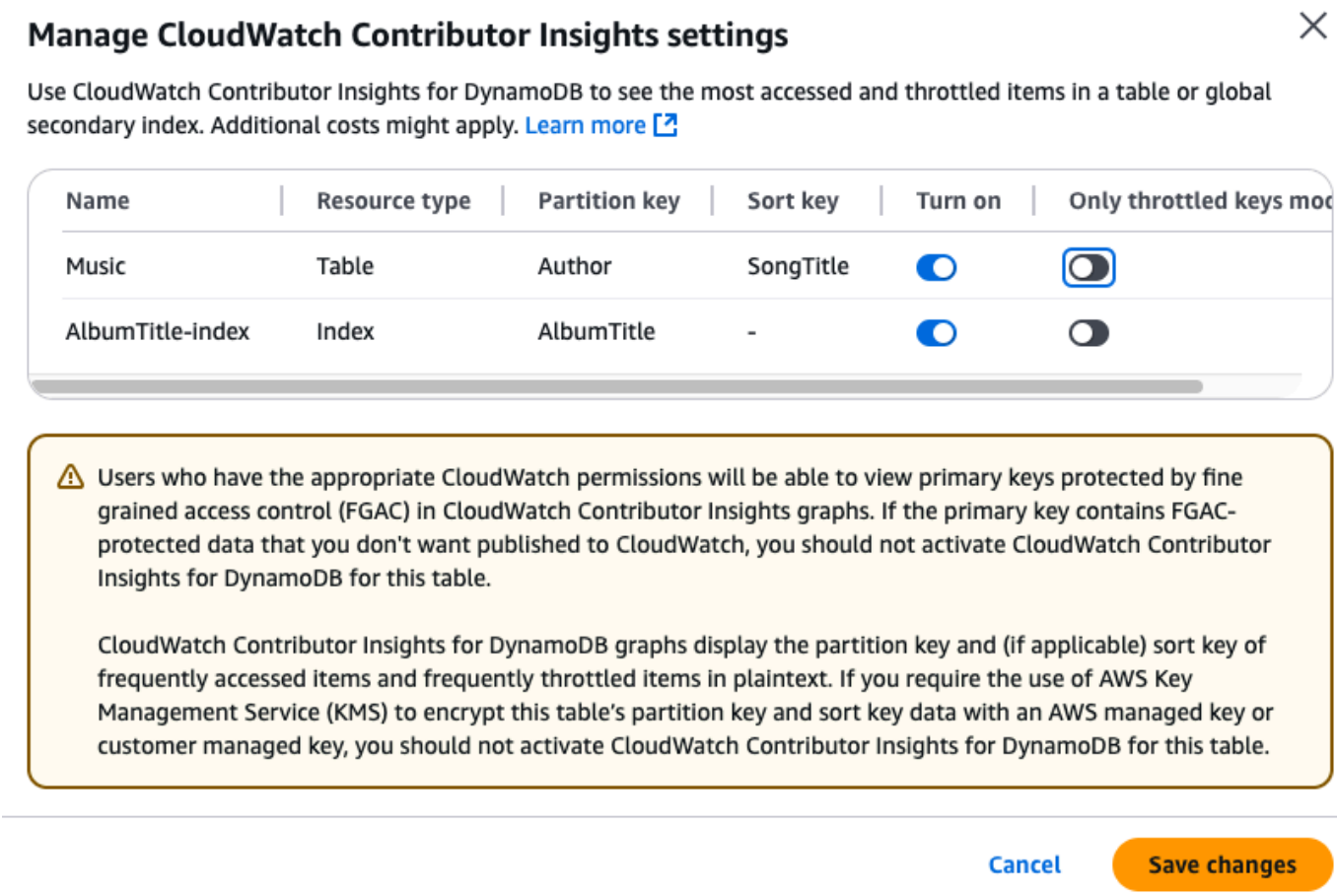
The console provides an intuitive way to enable Contributor Insights and select the appropriate mode for your monitoring needs.

To use Contributor Insights in the console

1. Sign in to the AWS Management Console and open the DynamoDB console at https://console.aws.amazon.com/dynamodb/.
2. In the navigation pane on the left side of the console, choose **Tables**.
3. Choose the `Music` table.
4. Choose the **Monitor** tab.
5. Choose **Turn on CloudWatch Contributor Insights**.



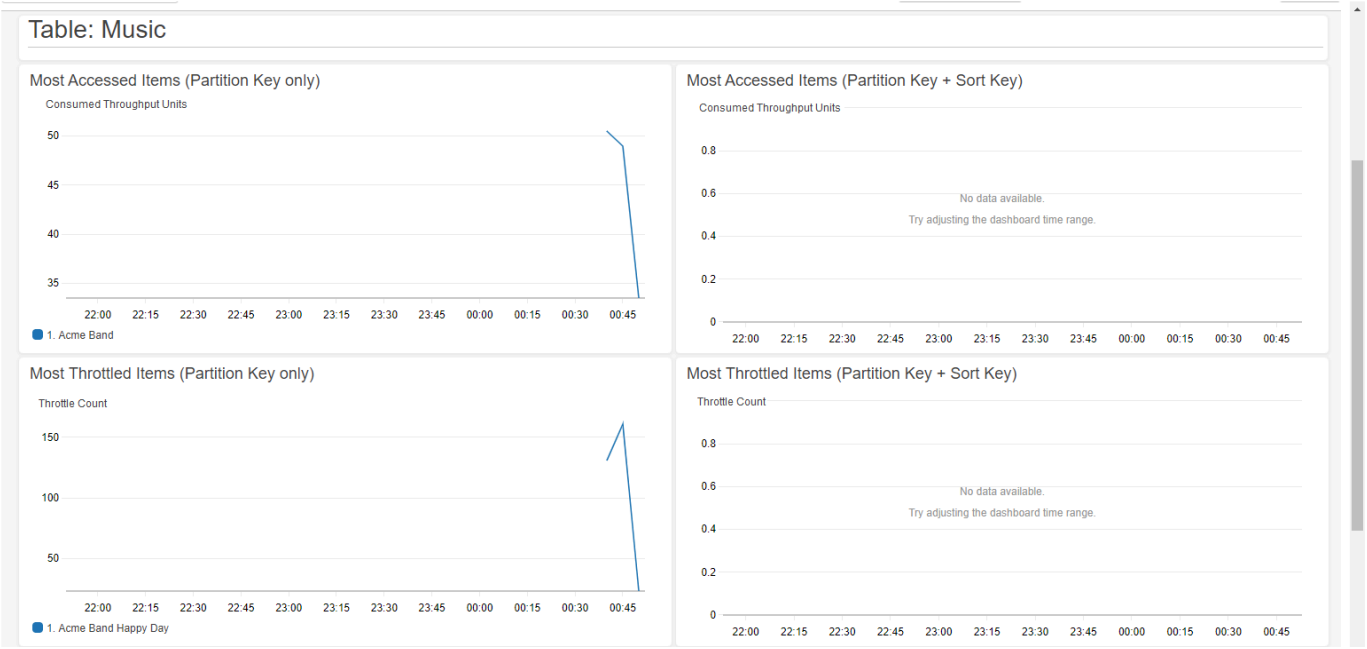
6. In the **Manage CloudWatch Contributor Insights settings** dialog box, toggle **Turn on** for both the `Music` base table and the `AlbumTitle-index` global secondary index.
7. Leave the **Only throttled keys mode** toggle in the off position for both and then choose **Save changes**.



This enables the default *accessed and throttled keys* mode for both the table and GSI, which provides monitoring of both accessed and throttled items. Switching the **Only throttled keys mode** toggle to the on position would enable the *throttled keys* mode.

If the operation fails, see DescribeContributorInsights FailureException in the *Amazon DynamoDB API Reference* for possible reasons.

8. The CloudWatch Contributor Insights graphs are now visible on the **Monitor** tab for the `Music` table. Since you enabled *accessed and throttled keys* mode, you see both accessed and throttled item graphs.



Switching between modes

You can switch between modes at any time without disabling Contributor Insights.

To switch Contributor Insights modes

1. On the **Monitor** tab of your table, choose **Manage CloudWatch Contributor Insights**.
2. In the **Manage Contributor Insights settings** dialog box, for each base table or GSIs:
   - Toggle **Only throttled keys mode** on or off to enable the *throttled keys* mode or go back to the default *accessed and throttled keys* mode.
   - Toggle **Turn on** off to disable CloudWatch Contributor Insight for a table or GSI.
3. Choose **Save changes**.

   Once complete, the graphs will reflect the new mode.

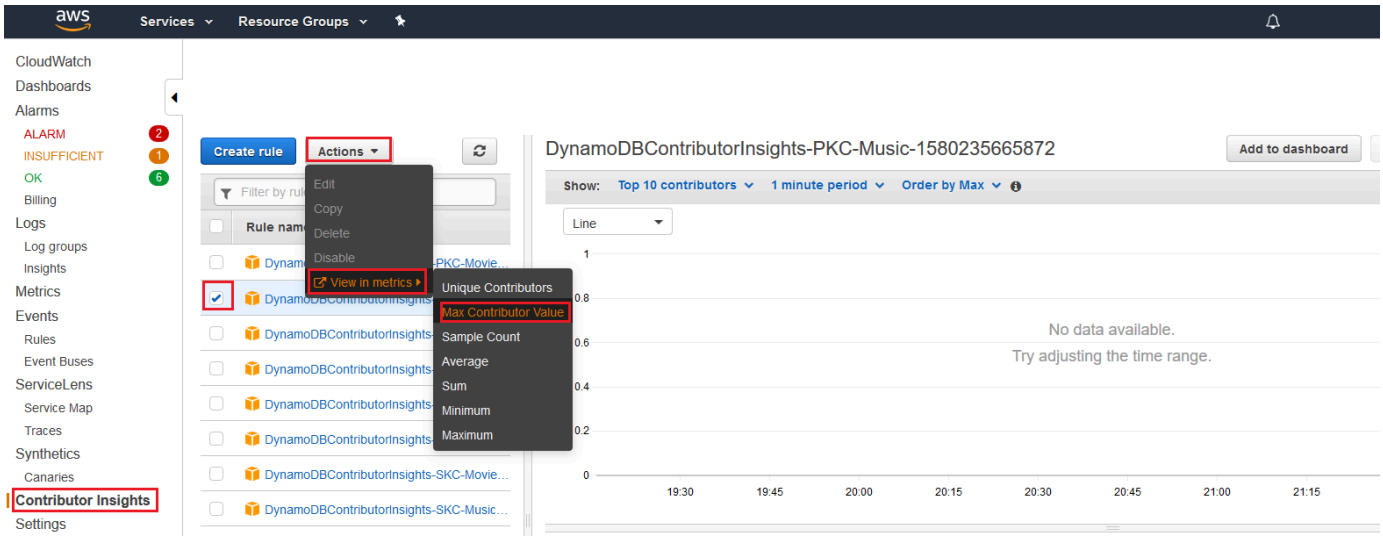
Creating CloudWatch alarms

Follow these steps to create a CloudWatch alarm and be notified when any partition key consumes more than 50,000 ConsumedThroughputUnits or experiences throttling.

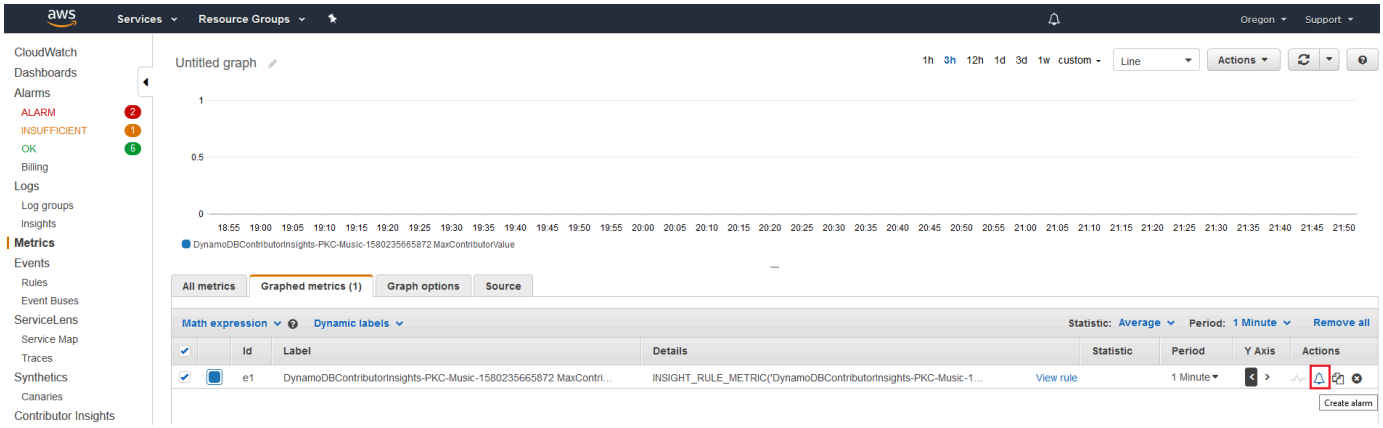
1. Sign in to the AWS Management Console and open the CloudWatch console at https://console.aws.amazon.com/cloudwatch/

2. In the navigation pane on the left side of the console, choose **Contributor Insights**.
3. Choose the appropriate rule based on your mode and what you want to monitor:
   - For accessed items monitoring (accessed and throttled keys mode only): Choose **DynamoDBContributorInsights-PKC-Music**
   - For throttled items monitoring (both modes): Choose **DynamoDBContributorInsights-PKT-Music**
4. Choose the **Actions** drop down.
5. Choose **View in metrics**.
6. Choose **Max Contributor Value**.

> **Note**
>
> Only `Max Contributor Value` and `Maximum` return useful statistics. The other statistics in this list don't return meaningful values.

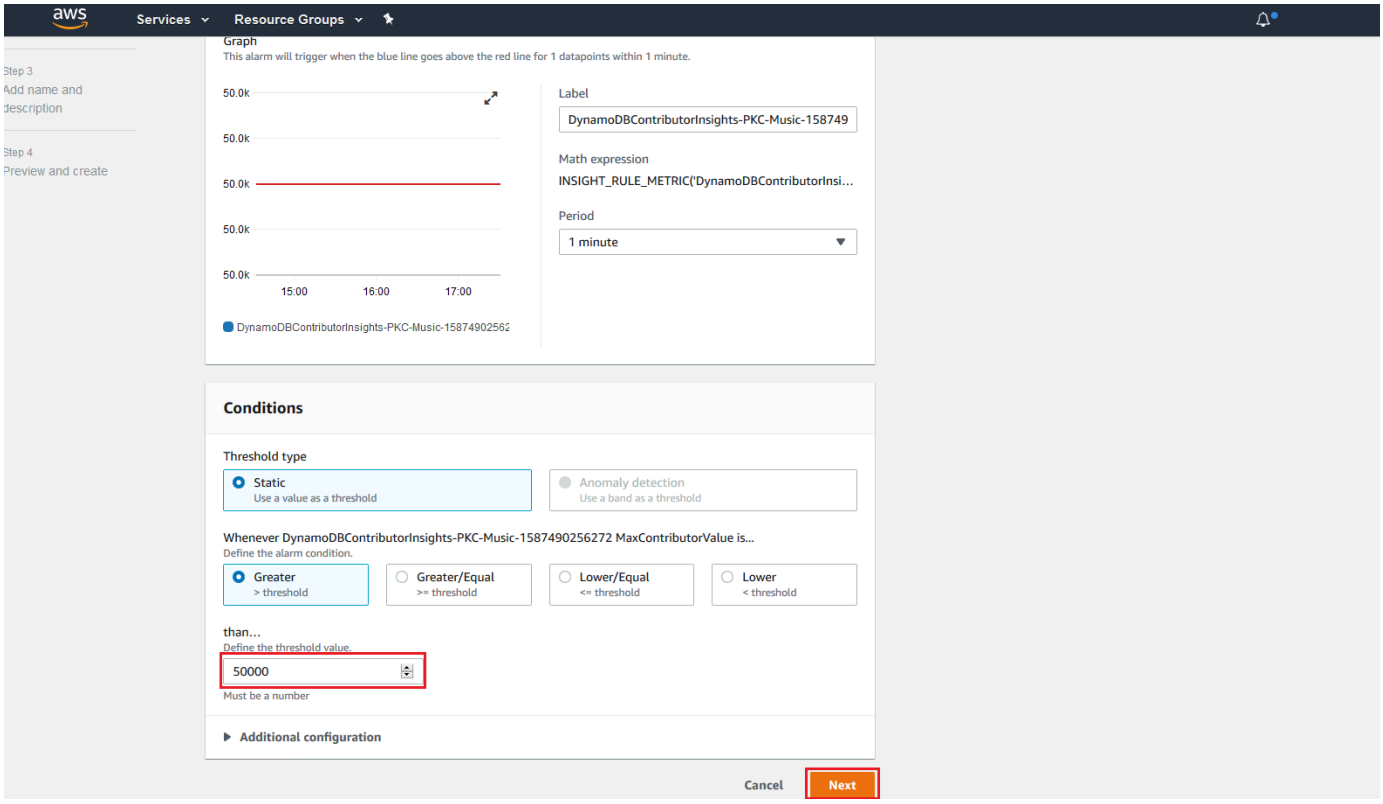


7. On the **Actions** column, Choose **Create Alarm**.



8. Enter an appropriate threshold value and choose **Next**:

   - For accessed items (PKC rules): Enter 50000 for `ConsumedThroughputUnits`
   - For throttled items (PKT rules): Enter 1 for `ThrottleCount` to be alerted on any throttling



9. See Using Amazon CloudWatch alarms for details on how to configure the notification for the alarm.

Using Contributor Insights (AWS CLI)

The AWS CLI provides programmatic access to Contributor Insights with full support for both modes. You can specify the mode when enabling Contributor Insights or switch modes later.

Basic operations with default mode

To use Contributor Insights with default settings

1. Enable CloudWatch Contributor Insights for DynamoDB on the `Music` base table with the *accessed and throttled keys* mode. Since `ACCESSED_AND_THROTTLED_KEYS` is the default mode, you can omit the `--contributor-insights-mode=ACCESSED_AND_THROTTLED_KEYS` parameter.

   ```
   aws dynamodb update-contributor-insights \
                       --table-name Music \
                       --contributor-insights-action=ENABLE
   ```

2. Enable Contributor Insights for DynamoDB on the `AlbumTitle-index` global secondary index.

   ```
   aws dynamodb update-contributor-insights \
                       --table-name Music \
                       --index-name AlbumTitle-index \
                       --contributor-insights-action=ENABLE
   ```

3. Get the status and rules for the `Music` table and all its indexes.

   ```
   aws dynamodb describe-contributor-insights
                       --table-name Music
   ```

   The response will include the `ContributorInsightsMode` field showing `ACCESSED_AND_THROTTLED_KEYS`.

4. List the status of the `Music` table and all its indexes.

   ```
   aws dynamodb list-contributor-insights --table-name Music
   ```

Enabling throttled keys mode

To enable Contributor Insights in throttled keys mode

1. Enable CloudWatch Contributor Insights for DynamoDB on the `Music` base table with *throttled keys* mode.

```
aws dynamodb update-contributor-insights \
    --table-name Music \
    --contributor-insights-action=ENABLE \
    --contributor-insights-mode=THROTTLED_KEYS
```

2. Enable Contributor Insights in *throttled keys* mode for the `AlbumTitle-index` global secondary index.

```
aws dynamodb update-contributor-insights \
    --table-name Music \
    --index-name AlbumTitle-index \
    --contributor-insights-action=ENABLE \
    --contributor-insights-mode=THROTTLED_KEYS
```

3. Verify the mode by describing the Contributor Insights configuration.

```
aws dynamodb describe-contributor-insights --table-name Music
```

   The response will show `ContributorInsightsMode` as `THROTTLED_KEYS` and fewer rules compared to the default mode.

Switching between modes

To switch Contributor Insights modes

1. Switch from *throttled keys* mode to *accessed and throttled keys* mode.

```
aws dynamodb update-contributor-insights \
    --table-name Music \
    --contributor-insights-action=ENABLE \
    --contributor-insights-mode=ACCESSED_AND_THROTTLED_KEYS
```

2. Switch from *accessed and throttled keys* mode to *throttled keys* mode.

```
aws dynamodb update-contributor-insights \
    --table-name Music \
    --contributor-insights-action=ENABLE \
    --contributor-insights-mode=THROTTLED_KEYS
```

3. Check the status during the transition.

```
aws dynamodb describe-contributor-insights --table-name Music
```

During the mode switch, the `ContributorInsightsStatus` will show as `ENABLING`. Once complete, it will show as `ENABLED` with the new mode.

Managing Contributor Insights

To manage Contributor Insights settings

1. Disable CloudWatch Contributor Insights for DynamoDB on the `AlbumTitle-index` global secondary index.

```
aws dynamodb update-contributor-insights \
                    --table-name Music --index-name AlbumTitle-index \
                    --contributor-insights-action=DISABLE
```

2. List all Contributor Insights configurations in your account.

```
aws dynamodb list-contributor-insights
```

This shows all tables and indexes with Contributor Insights enabled, along with their modes.

3. Get detailed information about a specific configuration.

```
aws dynamodb describe-contributor-insights \
                    --table-name Music \
                    --index-name AlbumTitle-index
```

Example responses

Here are example responses showing the differences between modes:

Accessed and throttled keys mode response

```
{
    "TableName": "Music",
    "ContributorInsightsRuleList": [
        "DynamoDBContributorInsights-PKC-Music-1234567890123",
        "DynamoDBContributorInsights-PKT-Music-1234567890123",
        "DynamoDBContributorInsights-SKC-Music-1234567890123",
        "DynamoDBContributorInsights-SKT-Music-1234567890123"
    ],
    "ContributorInsightsStatus": "ENABLED",
    "ContributorInsightsMode": "ACCESSED_AND_THROTTLED_KEYS",
    "LastUpdateDateTime": "2024-01-15T10:30:00.000Z"
}
```

Throttled keys mode response

```
{
    "TableName": "Music",
    "ContributorInsightsRuleList": [
        "DynamoDBContributorInsights-PKT-Music-1234567890123",
        "DynamoDBContributorInsights-SKT-Music-1234567890123"
    ],
    "ContributorInsightsStatus": "ENABLED",
    "ContributorInsightsMode": "THROTTLED_KEYS",
    "LastUpdateDateTime": "2024-01-15T10:35:00.000Z"
}
```

Notice that throttled keys mode has fewer rules (only PKT and SKT), which corresponds to a more focused monitoring.

Using IAM with CloudWatch contributor insights for DynamoDB

The first time that you enable Amazon CloudWatch Contributor Insights for Amazon DynamoDB, DynamoDB automatically creates an AWS Identity and Access Management (IAM) service-linked role for you. This role, `AWSServiceRoleForDynamoDBCloudWatchContributorInsights`, allows DynamoDB to manage CloudWatch Contributor Insights rules on your behalf. Don't delete this service-linked role. If you delete it, all your managed rules will no longer be cleaned up when you delete your table or global secondary index.

For more information about service-linked roles, see Using service-linked roles in the *IAM User Guide*.

The following permissions are required:

- To enable or disable CloudWatch Contributor Insights for DynamoDB, you must have `dynamodb:UpdateContributorInsights` permission on the table or index.
- To view CloudWatch Contributor Insights for DynamoDB graphs, you must have `cloudwatch:GetInsightRuleReport` permission.
- To describe CloudWatch Contributor Insights for DynamoDB for a given DynamoDB table or index, you must have `dynamodb:DescribeContributorInsights` permission.
- To list CloudWatch Contributor Insights for DynamoDB statuses for each table and global secondary index, you must have `dynamodb:ListContributorInsights` permission.

Example: Enable or disable CloudWatch contributor insights for DynamoDB

The following IAM policy grants permissions to enable or disable CloudWatch Contributor Insights for DynamoDB.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "iam:CreateServiceLinkedRole",
            "Resource": "arn:aws:iam::*:role/aws-
service-role/contributorinsights.dynamodb.amazonaws.com/
AWSServiceRoleForDynamoDBCloudWatchContributorInsights",
            "Condition": {"StringLike": {"iam:AWSServiceName":
 "contributorinsights.dynamodb.amazonaws.com"}}
        },
        {
            "Effect": "Allow",
            "Action": [
                "dynamodb:UpdateContributorInsights"
            ],
            "Resource": "arn:aws:dynamodb:*:*:table/*"
```

```
        }
    ]
}
```

For tables encryped by KMS key, the user needs to have kms:Decrypt permissions in order to update Contributor Insights.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "iam:CreateServiceLinkedRole",
            "Resource": "arn:aws:iam::*:role/aws-
service-role/contributorinsights.dynamodb.amazonaws.com/
AWSServiceRoleForDynamoDBCloudWatchContributorInsights",
            "Condition": {
                "StringLike": {
                    "iam:AWSServiceName":
 "contributorinsights.dynamodb.amazonaws.com"
                }
            }
        },
        {
            "Effect": "Allow",
            "Action": [
                "dynamodb:UpdateContributorInsights"
            ],
            "Resource": "arn:aws:dynamodb:*:*:table/*"
        },
        {
            "Effect": "Allow",
            "Resource": "arn:aws:kms:*:*:key/*",
            "Action": [
                "kms:Decrypt"
            ]
        }
    ]
}
```

Example: Retrieve CloudWatch contributor insights rule report

The following IAM policy grants permissions to retrieve CloudWatch Contributor Insights rule report.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "cloudwatch:GetInsightRuleReport"
            ],
            "Resource": "arn:aws:cloudwatch:*:*:insight-rule/
DynamoDBContributorInsights*"
        }
    ]
}
```

Example: Selectively apply CloudWatch contributor insights for DynamoDB permissions based on resource

The following IAM policy grants permissions to allow the `ListContributorInsights` and `DescribeContributorInsights` actions and denies the `UpdateContributorInsights` action for a specific global secondary index.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "dynamodb:ListContributorInsights",
                "dynamodb:DescribeContributorInsights"
            ],
```

```
            "Resource": "*"
        },
        {
            "Effect": "Deny",
            "Action": [
                "dynamodb:UpdateContributorInsights"
            ],
            "Resource": "arn:aws:dynamodb:us-west-2:123456789012:table/Books/
index/Author-index"
        }
    ]
}
```

Using service-linked roles for CloudWatch Contributor Insights for DynamoDB

CloudWatch Contributor Insights for DynamoDB uses AWS Identity and Access Management (IAM) service-linked roles. A service-linked role is a unique type of IAM role that is linked directly to CloudWatch Contributor Insights for DynamoDB. Service-linked roles are predefined by CloudWatch Contributor Insights for DynamoDB and include all the permissions that the service requires to call other AWS services on your behalf.

A service-linked role makes setting up CloudWatch Contributor Insights for DynamoDB easier because you don't have to manually add the necessary permissions. CloudWatch Contributor Insights for DynamoDB defines the permissions of its service-linked roles, and unless defined otherwise, only CloudWatch Contributor Insights for DynamoDB can assume its roles. The defined permissions include the trust policy and the permissions policy, and that permissions policy cannot be attached to any other IAM entity.

For information about other services that support service-linked roles, see AWS Services That Work with IAM and look for the services that have **Yes** in the **Service-Linked Role** column. Choose a **Yes** with a link to view the service-linked role documentation for that service.

Service-linked role permissions for CloudWatch Contributor Insights for DynamoDB

CloudWatch Contributor Insights for DynamoDB uses the service-linked role named **AWSServiceRoleForDynamoDBCloudWatchContributorInsights**. The purpose of the service-linked role is to allow Amazon DynamoDB to manage Amazon CloudWatch Contributor Insights rules created for DynamoDB tables and global secondary indexes, on your behalf.

The `AWSServiceRoleForDynamoDBCloudWatchContributorInsights` service-linked role trusts the following services to assume the role:

- `contributorinsights.dynamodb.amazonaws.com`

The role permissions policy allows CloudWatch Contributor Insights for DynamoDB to complete the following actions on the specified resources:

- Action: `Create and manage Insight Rules` on `DynamoDBContributorInsights`

You must configure permissions to allow an IAM entity (such as a user, group, or role) to create, edit, or delete a service-linked role. For more information, see Service-Linked Role Permissions in the *IAM User Guide*.

Creating a service-linked role for CloudWatch Contributor Insights for DynamoDB

You don't need to manually create a service-linked role. When you enable Contributor Insights in the AWS Management Console, the AWS CLI, or the AWS API, CloudWatch Contributor Insights for DynamoDB creates the service-linked role for you.

If you delete this service-linked role, and then need to create it again, you can use the same process to recreate the role in your account. When you enable Contributor Insights, CloudWatch Contributor Insights for DynamoDB creates the service-linked role for you again.

Editing a service-linked role for CloudWatch Contributor Insights for DynamoDB

CloudWatch Contributor Insights for DynamoDB does not allow you to edit the `AWSServiceRoleForDynamoDBCloudWatchContributorInsights` service-linked role. After you create a service-linked role, you cannot change the name of the role because various entities might reference the role. However, you can edit the description of the role using IAM. For more information, see Editing a service-linked role in the *IAM User Guide*.

Deleting a service-linked role for CloudWatch Contributor Insights for DynamoDB

You don't need to manually delete the `AWSServiceRoleForDynamoDBCloudWatchContributorInsights` role. When you disable Contributor Insights in the AWS Management Console, the AWS CLI, or the AWS API, CloudWatch Contributor Insights for DynamoDB cleans up the resources.

You can also use the IAM console, the AWS CLI or the AWS API to manually delete the service-linked role. To do this, you must first manually clean up the resources for your service-linked role and then you can manually delete it.

> **Note**
>
> If the CloudWatch Contributor Insights for DynamoDB service is using the role when you try to delete the resources, then the deletion might fail. If that happens, wait for a few minutes and try the operation again.

To manually delete the service-linked role using IAM

Use the IAM console, the AWS CLI, or the AWS API to delete the `AWSServiceRoleForDynamoDBCloudWatchContributorInsights` service-linked role. For more information, see Deleting a service-linked role in the *IAM User Guide*.

Best practices for designing and architecting with DynamoDB

Use this section to quickly find recommendations for maximizing performance and minimizing throughput costs when working with DynamoDB.

Topics

- NoSQL design for DynamoDB
- Using the DynamoDB Well-Architected Lens to optimize your DynamoDB workload
- Best practices for designing and using partition keys effectively in DynamoDB
- Best practices for using sort keys to organize data in DynamoDB
- Best practices for using secondary indexes in DynamoDB
- Best practices for storing large items and attributes in DynamoDB
- Best practices for handling time series data in DynamoDB
- Best practices for managing many-to-many relationships in DynamoDB tables
- Best practices for querying and scanning data in DynamoDB
- Best practices for DynamoDB table design
- Using DynamoDB global tables
- Best practices for managing the control plane in DynamoDB
- Best practices for using bulk data operations in DynamoDB
- Best practices for handling concurrent updates in DynamoDB
- Best practices for understanding your AWS billing and usage reports in DynamoDB
- Migrating a DynamoDB table from one account to another
- Prescriptive guidance to integrate DAX with DynamoDB applications

- Considerations when using AWS PrivateLink for Amazon DynamoDB

NoSQL design for DynamoDB

NoSQL database systems like Amazon DynamoDB use alternative models for data management, such as key-value pairs or document storage. When you switch from a relational database

management system to a NoSQL database system like DynamoDB, it's important to understand the key differences and specific design approaches.

Topics

- Differences between relational data design and NoSQL
- Two key concepts for NoSQL design
- Approaching NoSQL design
- NoSQL Workbench for DynamoDB

Differences between relational data design and NoSQL

Relational database systems (RDBMS) and NoSQL databases have different strengths and weaknesses:

- In RDBMS, data can be queried flexibly, but queries are relatively expensive and don't scale well in high-traffic situations (see First steps for modeling relational data in DynamoDB).
- In a NoSQL database such as DynamoDB, data can be queried efficiently in a limited number of ways, outside of which queries can be expensive and slow.

These differences make database design different between the two systems:

- In RDBMS, you design for flexibility without worrying about implementation details or performance. Query optimization generally doesn't affect schema design, but normalization is important.
- In DynamoDB, you design your schema specifically to make the most common and important queries as fast and as inexpensive as possible. Your data structures are tailored to the specific requirements of your business use cases.

Two key concepts for NoSQL design

NoSQL design requires a different mindset than RDBMS design. For an RDBMS, you can go ahead and create a normalized data model without thinking about access patterns. You can then extend it later when new questions and query requirements arise. You can organize each type of data into its own table.

How NoSQL design is different

- By contrast, you shouldn't start designing your schema for DynamoDB until you know the questions it will need to answer. Understanding the business problems and the application use cases up front is essential.
- You should maintain as few tables as possible in a DynamoDB application. Having fewer tables keeps things more scalable, requires less permissions management, and reduces overhead for your DynamoDB application. It can also help keep backup costs lower overall.

Approaching NoSQL design

The first step in designing your DynamoDB application is to identify the specific query patterns that the system must satisfy.

In particular, it is important to understand three fundamental properties of your application's access patterns before you begin:

- **Data size**: Knowing how much data will be stored and requested at one time will help determine the most effective way to partition the data.
- **Data shape**: Instead of reshaping data when a query is processed (as an RDBMS system does), a NoSQL database organizes data so that its shape in the database corresponds with what will be queried. This is a key factor in increasing speed and scalability.
- **Data velocity**: DynamoDB scales by increasing the number of physical partitions that are available to process queries, and by efficiently distributing data across those partitions. Knowing in advance what the peak query loads will be might help determine how to partition data to best use I/O capacity.

After you identify specific query requirements, you can organize data according to general principles that govern performance:

- **Keep related data together.** Research has shown that the principle of 'locality of reference', keeping related data together in one place, is a key factor in improving performance and response times in NoSQL systems, just as it was found to be important for optimizing routing tables many years ago.

  As a general rule, you should maintain as few tables as possible in a DynamoDB application.

Exceptions are cases where high-volume time series data are involved, or datasets that have very different access patterns. A single table with inverted indexes can usually enable simple queries to create and retrieve the complex hierarchical data structures required by your application.

- **Use sort order.** Related items can be grouped together and queried efficiently if their key design causes them to sort together. This is an important NoSQL design strategy.
- **Distribute queries.** It's also important that a high volume of queries not be focused on one part of the database, where they can exceed I/O capacity. Instead, you should design data keys to distribute traffic evenly across partitions as much as possible, avoiding hot spots.
- **Use global secondary indexes.** By creating specific global secondary indexes, you can enable different queries than your main table can support, and that are still fast and relatively inexpensive.

These general principles translate into some common design patterns that you can use to model data efficiently in DynamoDB.

NoSQL Workbench for DynamoDB

NoSQL Workbench for DynamoDB is a cross-platform, client-side GUI application that you can use for modern database development and operations. It's available for Windows, macOS, and Linux. NoSQL Workbench is a visual development tool that provides data modeling, data visualization, sample data generation, and query development features to help you design, create, query, and manage DynamoDB tables. With NoSQL Workbench for DynamoDB, you can build new data models from, or design models based on, existing data models that satisfy your application's data access patterns. You can also import and export the designed data model at the end of the process. For more information, see Building data models with NoSQL Workbench.

Using the DynamoDB Well-Architected Lens to optimize your DynamoDB workload

This section describes the Amazon DynamoDB Well-Architected Lens, a collection of design principles and guidance for designing well-architected DynamoDB workloads.

Optimizing costs on DynamoDB tables

This section covers best practices on how to optimize costs for your existing DynamoDB tables. You should look at the following strategies to see which cost optimization strategy best suits your

needs and approach them iteratively. Each strategy will provide an overview of what might be impacting your costs, what signs to look for, and prescriptive guidance on how to reduce them.

Topics

- Evaluate your costs at the table level
- Evaluate your DynamoDB table's capacity mode
- Evaluate your DynamoDB table's auto scaling settings
- Evaluate your DynamoDB table class selection
- Identify your unused resources in DynamoDB
- Evaluate your DynamoDB table usage patterns
- Evaluate your DynamoDB streams usage
- Evaluate your provisioned capacity for right-sized provisioning in your DynamoDB table

Evaluate your costs at the table level

The Cost Explorer tool found within the AWS Management Console allows you to see costs broken down by type, such as read, write, storage and backup charges. You can also see these costs summarized by period such as month or day.

One challenge administrators can face is when the costs of only one particular table need to be reviewed. Some of this data is available via the DynamoDB console or via calls to the `DescribeTable` API, however Cost Explorer does not, by default, allow you to filter or group by costs associated with a specific table. This section will show you how to use tagging to perform individual table cost analysis in Cost Explorer.

Topics

- How to view the costs of a single DynamoDB table
- Cost Explorer's default view
- How to use and apply table tags in Cost Explorer

How to view the costs of a single DynamoDB table

Both the Amazon DynamoDB AWS Management Console and the `DescribeTable` API will show you information about a single table, including the primary key schema, any indexes on the table,

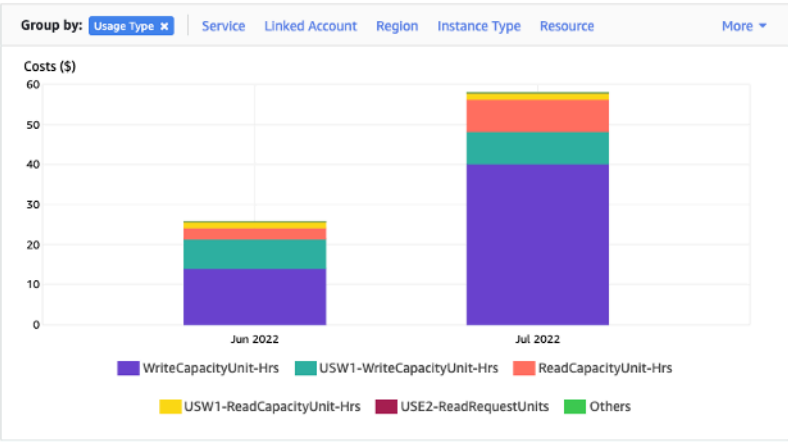
and the size and item count of the table and any indexes. The size of the table, plus the size of the indexes, can be used to calculate the monthly storage cost for your table. For example, $0.25 per GB in the us-east-1 region.

If the table is in provisioned capacity mode, the current RCU and WCU settings are returned as well. These could be used to calculate the current read and write costs for the table, but these costs could change, especially if the table has been configured with Auto Scaling.

> **Note**
>
> If the table is in on-demand capacity mode, then `DescribeTable` will not help estimate throughput costs, as these are billed based on actual, not provisioned usage in any one period.

Cost Explorer's default view

Cost Explorer's default view provides charts showing the cost of consumed resources such as throughput and storage. You can choose to group costs by period, such as totals by month or by day. The costs of storage, reads, writes, and other features can be broken out and compared as well.



How to use and apply table tags in Cost Explorer

By default, Cost Explorer does not provide a summary of the costs for any one specific table, as it will combine the costs of multiple tables into a total. However, you can use AWS resource tagging to identify each table by a metadata tag. Tags are key-value pairs you can use for a variety of purposes, such as to identify all resources belonging to a project or department. For this example, we'll assume you have a table named **MyTable**.

1. Set a tag with the key of **table_name** and the value of **MyTable**.
2. Activate the tag within Cost Explorer and then filter on the tag value to gain more visibility into each table's costs.

> **Note**
>
> It may take one or two days for the tag to start appearing in Cost Explorer

You can set metadata tags yourself in the console, or via automation such as the AWS CLI or AWS SDK. Consider requiring a **table_name** tag to be set as part of your organization's new table creation process. For existing tables, there is a Python utility available that will find and apply these tags to all existing tables in a certain region in your account. See Eponymous Table Tagger on GitHub for more details.

Evaluate your DynamoDB table's capacity mode

This section provides an overview of how to select the appropriate capacity mode for your DynamoDB table. Each mode is tuned to meet the needs of a different workload in terms of responsiveness to change in throughput, as well as how that usage is billed. You must balance these factors when making your decision.

Topics

- What table capacity modes are available
- When to select on-demand capacity mode
- When to select provisioned capacity mode
- Additional factors to consider when choosing a table capacity mode

What table capacity modes are available

When you create a DynamoDB table, you must select either on-demand or provisioned capacity mode.

You can switch tables from provisioned capacity mode to on-demand mode up to four times in a 24-hour rolling window. You can switch tables from on-demand mode to provisioned capacity mode at any time.

On-demand capacity mode

The on-demand capacity mode is designed to eliminate the need to plan or provision the capacity of your DynamoDB table. In this mode, your table will instantly accommodate requests to your table without the need to scale any resources up or down (up to twice the previous peak throughput of the table).

DynamoDB on-demand offers pay-per-request pricing for read and write requests so that you only pay for what you use.

Provisioned capacity mode

The provisioned capacity mode is a more traditional model where you must define how much capacity the table has available for requests either directly or with the assistance of auto scaling. Because a specific capacity is provisioned for the table at any given time, billing is based off of the total capacity provisioned rather than the number of requests consumed. Going over the allocated capacity can also cause the table to reject requests and reduce the experience of your applications users.

Provisioned capacity mode requires constant monitoring to find a balance between not over-provisioning or under-provisioning the table to keep both throttling low and costs tuned.

When to select on-demand capacity mode

When optimizing for cost, on-demand mode is your best choice when you have a workload similar to the following graphs.

The following factors contribute to this type of workload:

- Traffic pattern that evolves over time
- Variable volume of requests (resulting from batch workloads)
- Unpredictable request timing (resulting in traffic spikes)
- Drops to zero or below 30% of the peak for a given hour

For workloads with the above factors, using auto scaling to maintain enough capacity on the table to respond to spikes in traffic will likely lead to the table being overprovisioned and costing more than necessary or the table being under provisioned and requests being unnecessarily throttled. On-demand capacity mode is the better choice because it can handle fluctuating traffic without requiring you to predict or adjust capacity.

With on-demand mode's pay-per-request pricing model, you don't have to worry about idle capacity because you only pay for the throughput you actually use. You are billed per read or write request consumed, so your costs directly reflect your actual usage, making it easy to balance costs and performance. Optionally, you can also configure maximum read or write (or both) throughput per second for individual on-demand tables and global secondary indexes to help keep costs and usage bounded. For more information, see maximum throughput for on-demand tables .

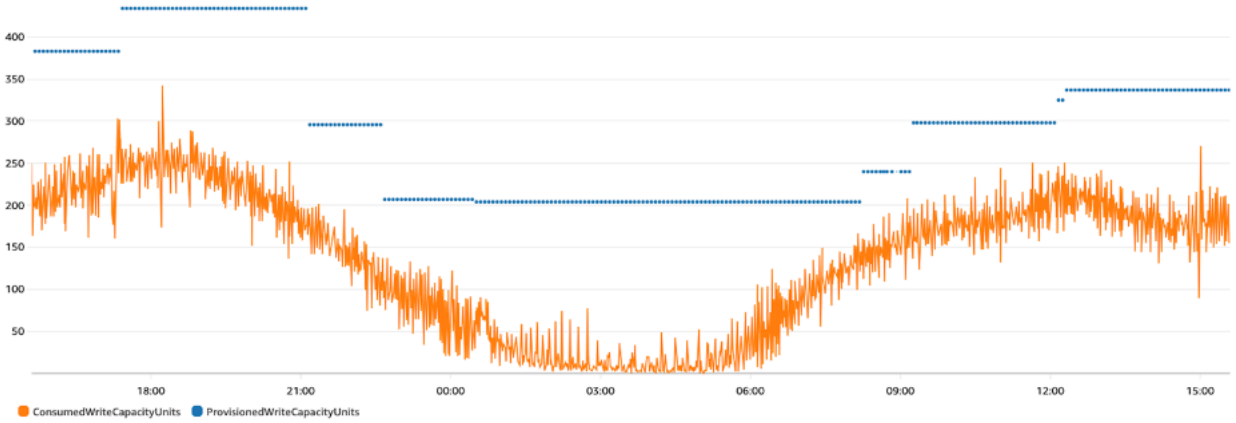
When to select provisioned capacity mode

An ideal workload for provisioned capacity mode is one with a more steady and predictable usage pattern like the graph below.

> **Note**
>
> We recommend reviewing metrics at a fine-grained period, such as 14 days, before taking action on your provisioned capacity.

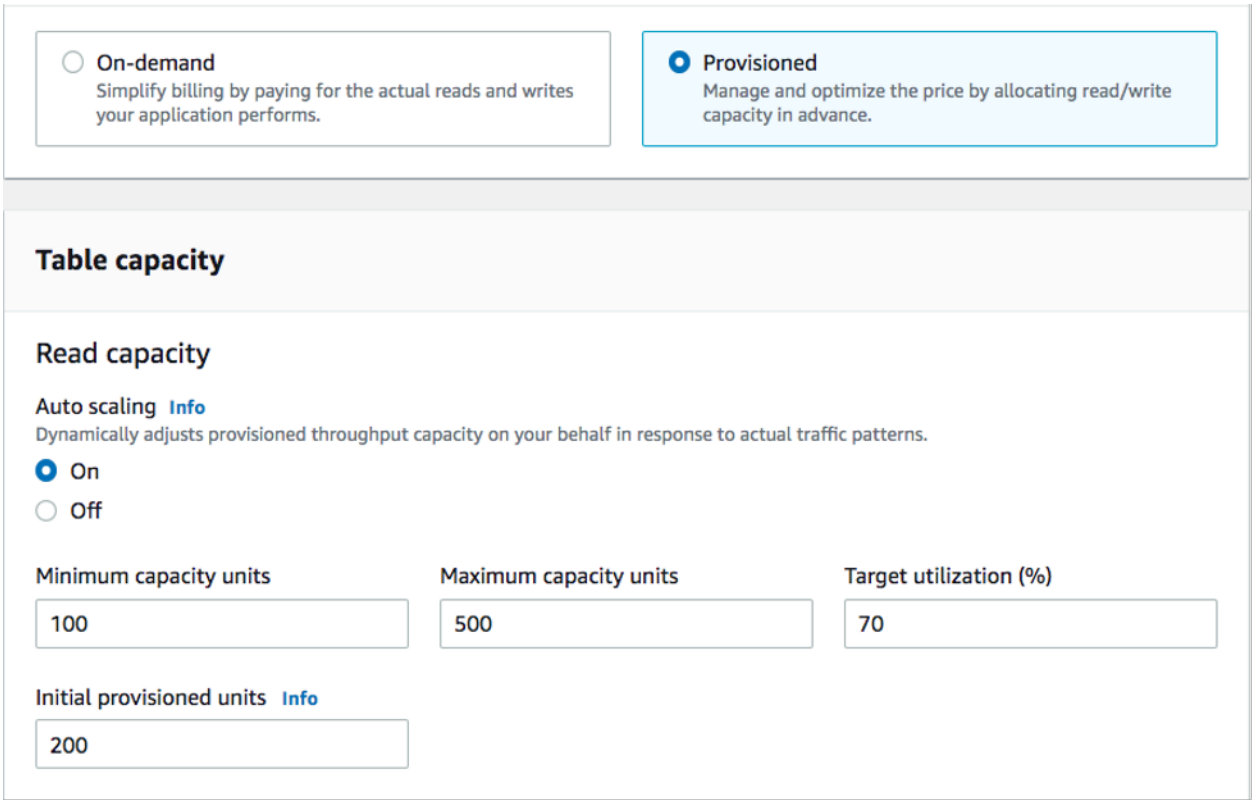
The following factors contribute to this type of workload:

- Steady, predictable and cyclical traffic for a given hour or day
- Limited short-term bursts of traffic



Since the traffic volumes within a given hour or day are more stable, you can set the provisioned capacity of the table relatively close to the actual consumed capacity of the table. Cost optimizing a provisioned capacity table is ultimately an exercise in getting the provisioned capacity (blue line) as close to the consumed capacity (orange line) as possible without increasing `ThrottledRequests` on the table. The space between the two lines is both wasted capacity as well as insurance against a bad user experience due to throttling. If you can predict your application's throughput requirements and you prefer the cost predictability of controlling read and write capacity, then you may want to continue using provisioned tables.

DynamoDB provides auto scaling for provisioned capacity tables which will automatically balance this on your behalf. This lets you track your consumed capacity throughout the day and set the capacity of the table based on a handful of variables. When using auto scaling, your table will be over-provisioned and you need to fine tune the ratio between number of throttles versus over-provisioned capacity units to match your workload needs.



Minimum capacity units

You can set the minimum capacity of a table to limit throttling, but it will not reduce the cost of the table. If your table has periods of low usage follow by a sudden burst of high usage, setting the minimum can prevent auto scaling from setting the table capacity too low.

Maximum capacity units

You can set the maximum capacity of a table to limit a table scaling higher than intended. Consider applying a maximum for Dev or Test tables where large-scale load testing is not desired. You can set a maximum for any table, but be sure to regularly evaluate this setting against the table baseline when using it in Production to prevent accidental throttling.

Target utilization

Setting the target utilization of the table is the primary means of cost optimization for a provisioned capacity table. Setting a lower percent value here will increase how much the table is overprovisioned, increasing cost, but reducing the risk of throttling. Setting a higher percent value will decrease how much the table is overprovisioned, but increase the risk of throttling.

Additional factors to consider when choosing a table capacity mode

When deciding between the two modes, there are some additional factors worth considering.

Provisioned capacity utilization

To determine when on-demand mode would cost less than provisioned capacity, it's helpful to look at your provisioned capacity utilization, which refers to how efficiently the allocated (or “provisioned) resources are being used. On-demand mode costs less for workloads with average provisioned capacity utilization below 35%. In many cases, even for workloads with provisioned capacity utilization higher than 35%, it can be more cost-effective to use on-demand mode especially if the workload has periods of low activity mixed with occasional peaks.

Reserved capacity

For provisioned capacity tables, DynamoDB offers the ability to purchase reserved capacity for your read and write capacity (replicated write capacity units (rWCU) and Standard-IA tables are currently not eligible). Reserved capacity offers significant discounts over standard provisioned capacity pricing.

When deciding between the two table modes, consider how much this additional discount will affect the cost of the table. In some cases, it may cost less to run a relatively unpredictable workload can be cheaper to run on an overprovisioned provisioned capacity table with reserved capacity.

Improving predictability of your workload

In some situations, a workload may seemingly have both a predictable and unpredictable pattern. While this can be easily supported with an on-demand table, costs will likely be better if the unpredictable patterns in the workload can be improved.

One of the most common causes of these patterns is batch imports. This type of traffic can often exceed the baseline capacity of the table to such a degree that throttling would occur if it were to run. To keep a workload like this running on a provisioned capacity table, consider the following options:

- If the batch occurs at scheduled times, you can schedule an increase to your auto- scaling capacity before it runs
- If the batch occurs randomly, consider trying to extend the time it runs rather than executing as fast as possible

- Add a ramp up period to the import where the velocity of the import starts small but is slowly increased over a few minutes until auto scaling has had the opportunity to start adjusting table capacity

Evaluate your DynamoDB table's auto scaling settings

This section provides an overview of how to evaluate the auto scaling settings on your DynamoDB tables. Amazon DynamoDB auto scaling is a feature that manages table and global secondary index (GSI) throughput based on your application traffic and your target utilization metric. This ensures your tables or GSIs will have the required capacity required for your application patterns.

The AWS auto scaling service will monitor your current table utilization and compare it to the target utilization value: `TargetValue`. It will notify you if it is time to increase or decrease the capacity allocated.

Topics

- Understanding your auto scaling settings
- How to identify tables with low target utilization (<=50%)
- How to address workloads with seasonal variance
- How to address spiky workloads with unknown patterns
- How to address workloads with linked applications

Understanding your auto scaling settings

Defining the correct value for the target utilization, initial step, and final values is an activity that requires involvement from your operations team. This allows you to properly define the values based on historical application usage, which will be used to trigger the AWS auto scaling policies. The utilization target is the percentage of your total capacity that needs to be hit during a period of time before the auto scaling rules apply.

When you set a **high utilization target (a target around 90%)** it means your traffic needs to be higher than 90% for a period of time before the auto scaling kicks in. You should not use a high utilization target unless your application is very constant and doesn't receive spikes in traffic.

When you set a very **low utilization (a target less than 50%)** it means your application would need to reach 50% of the provisioned capacity before it triggers an auto scaling policy. Unless your

application traffic grows at a very aggressive rate, this usually translates into unused capacity and wasted resources.

How to identify tables with low target utilization (<=50%)

You can use either the AWS CLI or AWS Management Console to monitor and identify the `TargetValues` for your auto scaling policies in your DynamoDB resources:

AWS CLI

1. Return the entire list of resources by running the following command:

   ```
   aws application-autoscaling describe-scaling-policies --service-namespace
    dynamodb
   ```

   This command will return the entire list of auto scaling policies that are issued to any DynamoDB resource. If you only want to retrieve the resources from a particular table, you can add the `–resource-id parameter`. For example:

   ```
   aws application-autoscaling describe-scaling-policies --service-namespace
    dynamodb --resource-id "table/<table-name>”
   ```

2. Return only the auto scaling policies for a particular GSI by running the following command

   ```
   aws application-autoscaling describe-scaling-policies --service-namespace
    dynamodb --resource-id "table/<table-name>/index/<gsi-name>”
   ```

   The values we're interested in for the auto scaling policies are highlighted below. We want to ensure that the target value is greater than 50% to avoid over-provisioning. You should obtain a result similar to the following:

   ```
   {
       "ScalingPolicies": [
           {
               "PolicyARN": "arn:aws:autoscaling:<region>:<account-
   id>:scalingPolicy:<uuid>:resource/dynamodb/table/<table-name>/index/<index-
   name>:policyName/$<full-gsi-name>-scaling-policy",
               "PolicyName": $<full-gsi-name>”,
               "ServiceNamespace": "dynamodb",
               "ResourceId": "table/<table-name>/index/<index-name>",
               "ScalableDimension": "dynamodb:index:WriteCapacityUnits",
   ```

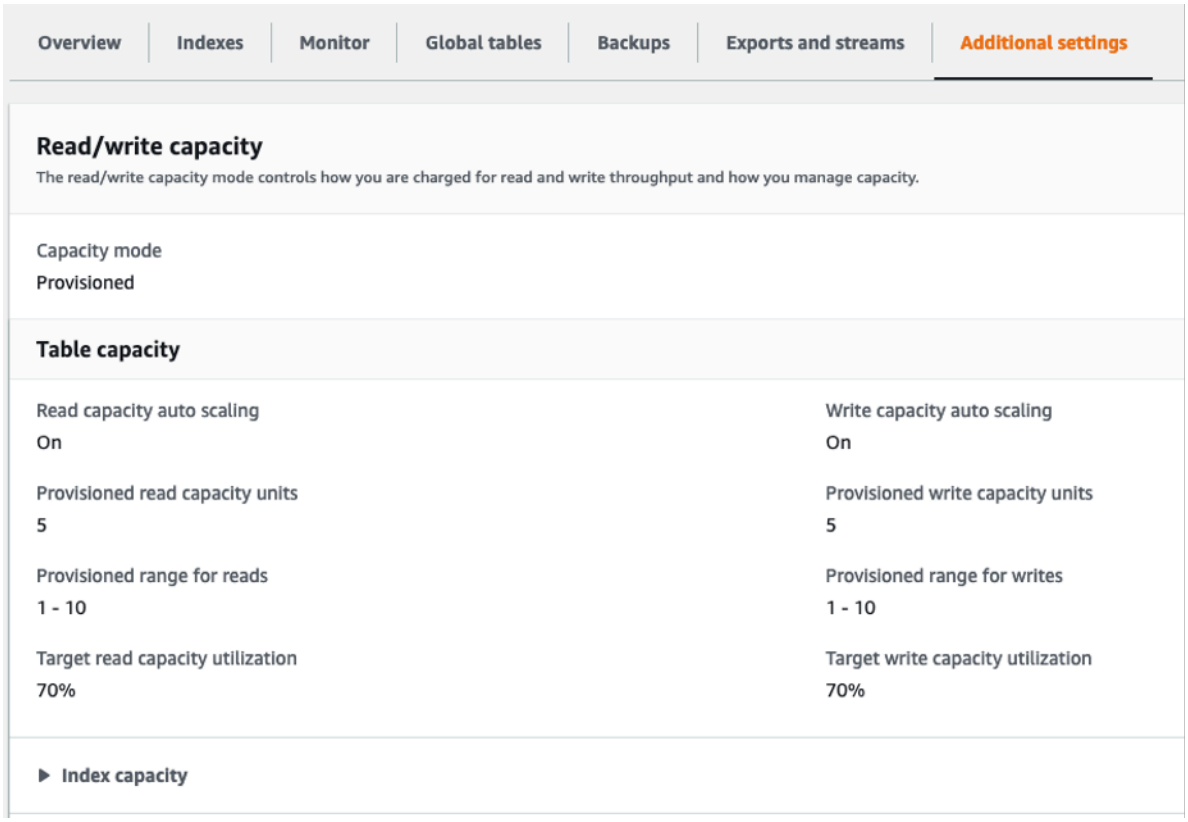
```
            "PolicyType": "TargetTrackingScaling",
            "TargetTrackingScalingPolicyConfiguration": {
                "TargetValue": 70.0,
                "PredefinedMetricSpecification": {
                    "PredefinedMetricType": "DynamoDBWriteCapacityUtilization"
                }
            },
            "Alarms": [
                ...
            ],
            "CreationTime": "2022-03-04T16:23:48.641000+10:00"
        },
        {
            "PolicyARN": "arn:aws:autoscaling:<region>:<account-
id>:scalingPolicy:<uuid>:resource/dynamodb/table/<table-name>/index/<index-
name>:policyName/$<full-gsi-name>-scaling-policy",
            "PolicyName":$<full-gsi-name>”,
            "ServiceNamespace": "dynamodb",
            "ResourceId": "table/<table-name>/index/<index-name>",
            "ScalableDimension": "dynamodb:index:ReadCapacityUnits",
            "PolicyType": "TargetTrackingScaling",
            "TargetTrackingScalingPolicyConfiguration": {
                "TargetValue": 70.0,
                "PredefinedMetricSpecification": {
                    "PredefinedMetricType": "DynamoDBReadCapacityUtilization"
                }
            },
            "Alarms": [
                ...
            ],
            "CreationTime": "2022-03-04T16:23:47.820000+10:00"
        }
    ]
}
```

AWS Management Console

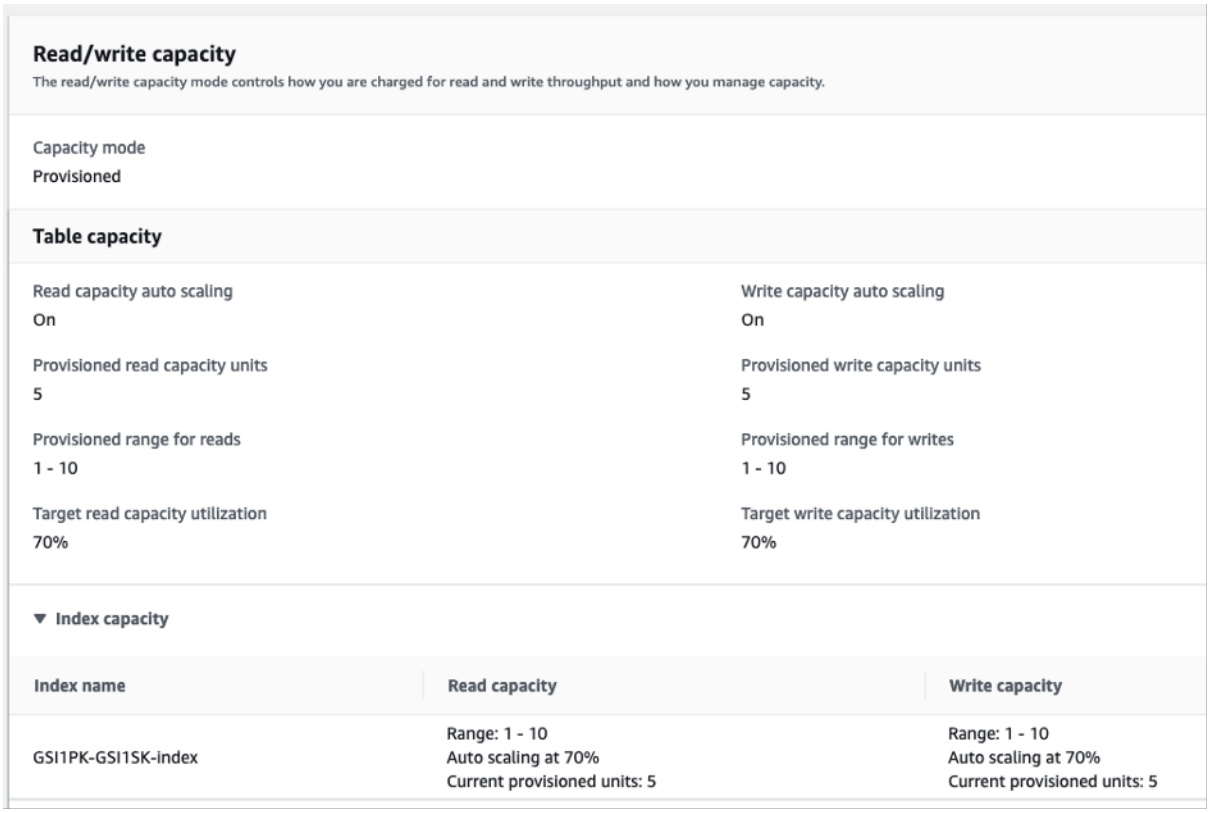
1. Sign in to the AWS Management Console and open the DynamoDB console at https://console.aws.amazon.com/dynamodb/.

   Select an appropriate AWS Region if necessary.

2. On the left navigation bar, select **Tables**. On the **Tables** page, select the table's **Name**.
3. On the *Table details* page, choose **Additional settings**, and then review your table's auto scaling settings.



For indexes, expand the **Index capacity** section to review the index's auto scaling settings.



If your target utilization values are less than or equal to 50%, you should explore your table utilization metrics to see if they are under-provisioned or over-provisioned.

How to address workloads with seasonal variance

Consider the following scenario: your application is operating under a minimum average value most of the time, but the utilization target is low so your application can react quickly to events that happen at certain hours in the day and you have enough capacity and avoid getting throttled. This scenario is common when you have an application that is very busy during normal office hours (9 AM to 5 PM) but then it works at a base level during after hours. Since some users will start to connect before 9 am, the application uses this low threshold to ramp up quickly to get to the *required* capacity during peak hours.

This scenario could look like this:

- Between 5 PM and 9 AM the `ConsumedWriteCapacity` units stay between 90 and 100
- Users start to connect to the application before 9 AM and the capacity units increases considerably (the maximum value you've seen is 1500 WCU)
- On average, your application usage varies between 800 to 1200 during working hours

If the previous scenario applies to you, consider using scheduled auto scaling, where your table could still have an application auto scaling rule configured, but with a less aggressive target utilization that only provisions the extra capacity at the specific intervals you require.

You can use AWS CLI to execute the following steps to create a scheduled auto scaling rule that will execute based on the time of day and the day of the week.

1. Register your DynamoDB table or GSI as scalable target with Application Auto Scaling. A scalable target is a resource that Application Auto Scaling can scale out or in.

```
aws application-autoscaling register-scalable-target \
    --service-namespace dynamodb \
    --scalable-dimension dynamodb:table:WriteCapacityUnits \
    --resource-id table/<table-name> \
    --min-capacity 90 \
    --max-capacity 1500
```

2. Set up scheduled actions according to your requirements.

   We'll need two rules to cover the scenario: one to scale up and another to scale down. The first rule to scale up the scheduled action:

```
aws application-autoscaling put-scheduled-action \
    --service-namespace dynamodb \
    --scalable-dimension dynamodb:table:WriteCapacityUnits \
    --resource-id table/<table-name> \
    --scheduled-action-name my-8-5-scheduled-action \
    --scalable-target-action MinCapacity=800,MaxCapacity=1500 \
    --schedule "cron(45 8 ? * MON-FRI *)" \
    --timezone "Australia/Brisbane"
```

   The second rule to scale down the scheduled action:

```
aws application-autoscaling put-scheduled-action \
    --service-namespace dynamodb \
    --scalable-dimension dynamodb:table:WriteCapacityUnits \
    --resource-id table/<table-name> \
    --scheduled-action-name my-5-8-scheduled-down-action \
    --scalable-target-action MinCapacity=90,MaxCapacity=1500 \
    --schedule "cron(15 17 ? * MON-FRI *)" \
    --timezone "Australia/Brisbane"
```

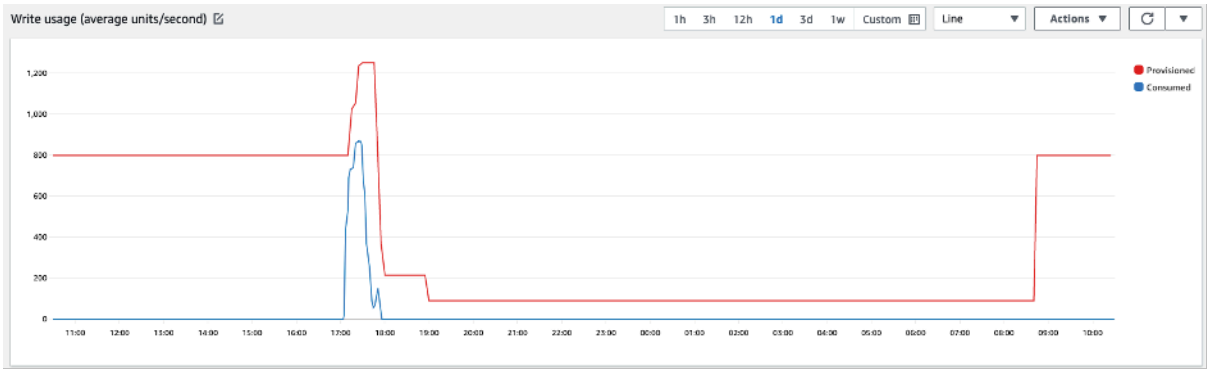
3. Run the following command to validate both rules have been activated:

```
aws application-autoscaling describe-scheduled-actions --service-namespace dynamodb
```

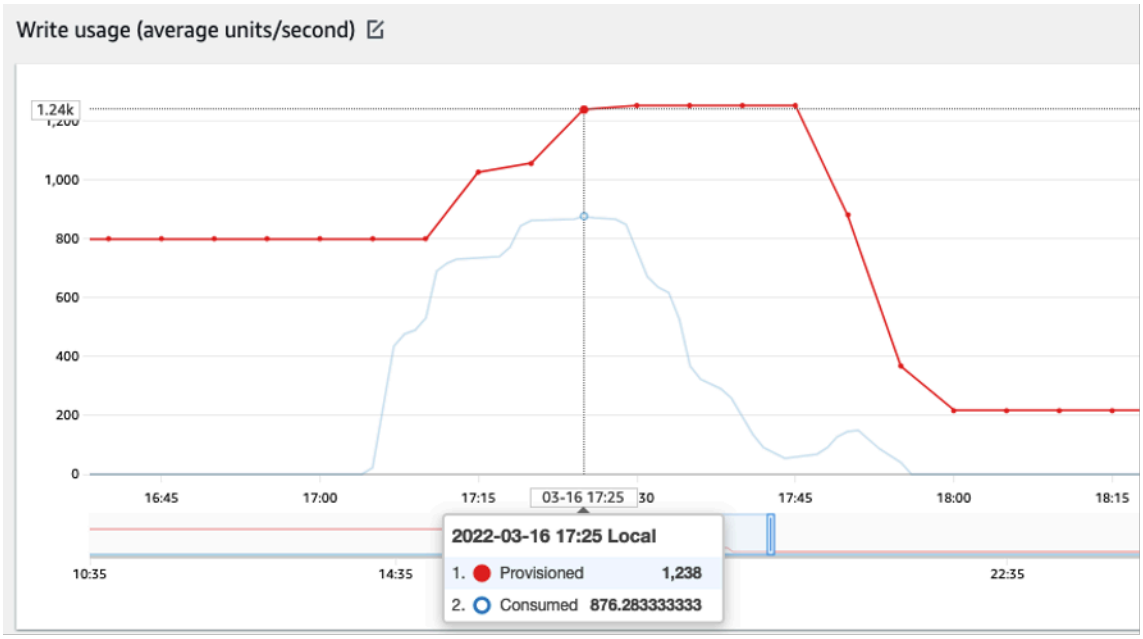
You should get a result like this:

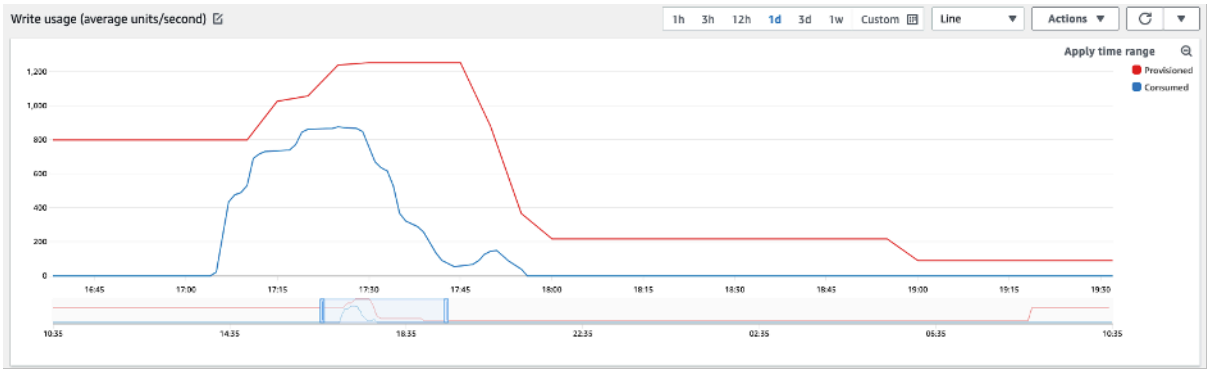
```
{
    "ScheduledActions": [
        {
            "ScheduledActionName": "my-5-8-scheduled-down-action",
            "ScheduledActionARN":
 "arn:aws:autoscaling:<region>:<account>:scheduledAction:<uuid>:resource/dynamodb/
table/<table-name>:scheduledActionName/my-5-8-scheduled-down-action",
            "ServiceNamespace": "dynamodb",
            "Schedule": "cron(15 17 ? * MON-FRI *)",
            "Timezone": "Australia/Brisbane",
            "ResourceId": "table/<table-name>",
            "ScalableDimension": "dynamodb:table:WriteCapacityUnits",
            "ScalableTargetAction": {
                "MinCapacity": 90,
                "MaxCapacity": 1500
            },
            "CreationTime": "2022-03-15T17:30:25.100000+10:00"
        },
        {
            "ScheduledActionName": "my-8-5-scheduled-action",
            "ScheduledActionARN":
 "arn:aws:autoscaling:<region>:<account>:scheduledAction:<uuid>:resource/dynamodb/
table/<table-name>:scheduledActionName/my-8-5-scheduled-action",
            "ServiceNamespace": "dynamodb",
            "Schedule": "cron(45 8 ? * MON-FRI *)",
            "Timezone": "Australia/Brisbane",
            "ResourceId": "table/<table-name>",
            "ScalableDimension": "dynamodb:table:WriteCapacityUnits",
            "ScalableTargetAction": {
                "MinCapacity": 800,
                "MaxCapacity": 1500
            },
            "CreationTime": "2022-03-15T17:28:57.816000+10:00"
        }
    ]
}
```

The following picture shows a sample workload that always keeps the 70% target utilization. Notice how the auto scaling rules are still applying and the throughput will not be reduced.



Zooming in, we can see there was a spike in the application that triggered the 70% auto scaling threshold, forcing the auto scaling to kick in and provide the extra capacity required for the table. The scheduled auto scaling action will affect maximum and minimum values, and it is your responsibility to set them up.





How to address spiky workloads with unknown patterns

In this scenario, the application uses a very low utilization target because you don't know the application patterns yet, and you want to ensure your workload is not throttled.

Consider using on-demand capacity mode instead. On-demand tables are perfect for spiky workloads where you don't know the traffic patterns. With on-demand capacity mode, you pay per request for the data reads and writes your application performs on your tables. You do not need to specify how much read and write throughput you expect your application to perform, as DynamoDB instantly accommodates your workloads as they ramp up or down.

How to address workloads with linked applications

In this scenario, the application depends on other systems, like batch processing scenarios where you can have big spikes in traffic according to events in the application logic.

Consider developing custom auto scaling logic that reacts to those events where you can increase table capacity and `TargetValues` depending on your specific needs. You could benefit from Amazon EventBridge and use a combination of AWS services like Lambda and Step Functions to react to your specific application needs.

Evaluate your DynamoDB table class selection

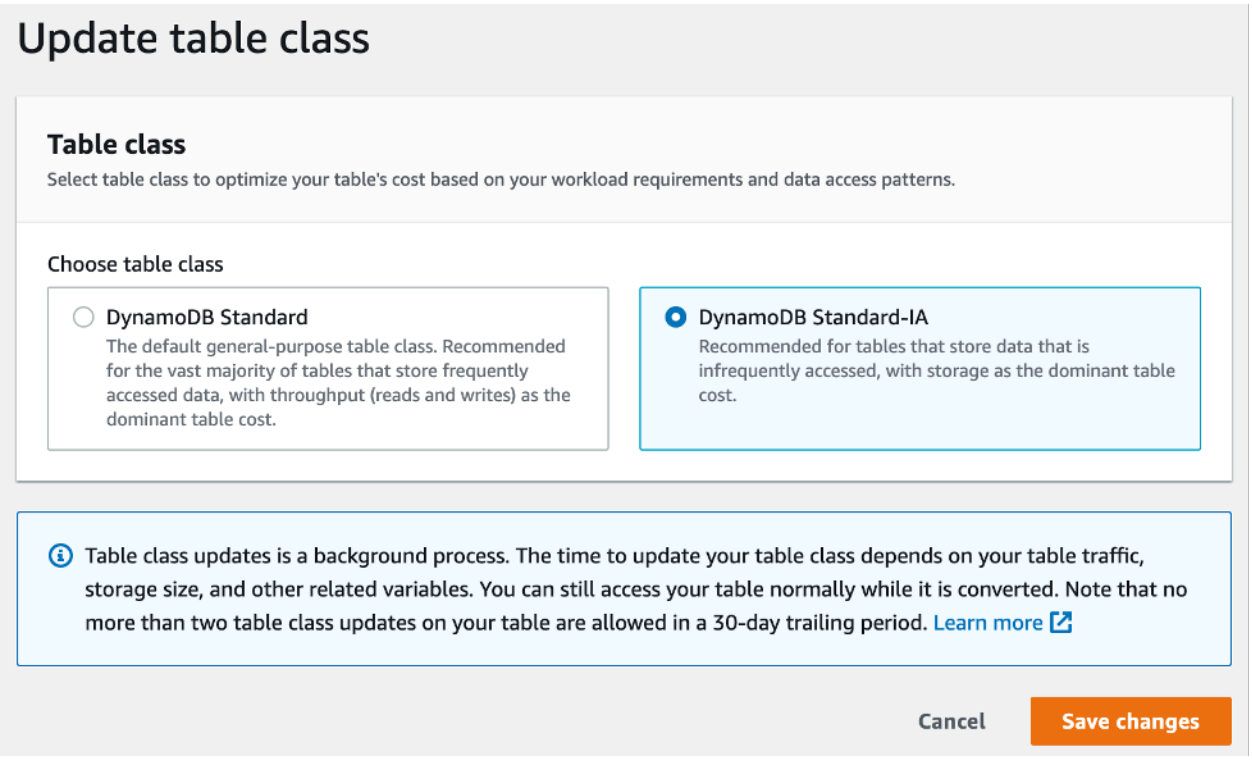
This section provides an overview of how to select the appropriate table class for your DynamoDB table. With the launch of the Standard Infrequent-Access (Standard-IA) table class, you now have the ability to optimize a table for either lower storage cost or lower throughput cost.

Topics

- What table classes are available
- When to select the DynamoDB Standard table class
- When to select DynamoDB Standard-IA table class
- Additional factors to consider when choosing a table class

What table classes are available

When you create a DynamoDB Table, you must select either DynamoDB Standard or DynamoDB Standard-IA for the table class. The table class can be changed twice in a 30-day period, so you can always change it in the future. Selecting either table class has no effect on table performance, availability, reliability, or durability.



Standard table class

The Standard table class is the default option for new tables. This option maintains the original billing balance of DynamoDB which offers a balance of throughput and storage costs for tables with frequently accessed data.

Standard-IA table class

The Standard-IA table class offers a lower storage cost (~60% lower) for workloads that require long-term storage of data with infrequent updates or reads. Since the class is optimized for infrequent access, reads and writes will be billed at a slightly higher cost (~25% higher) than the Standard table class.

When to select the DynamoDB Standard table class

DynamoDB Standard table class is best suited for tables whose storage cost is approximately 50% or less of the overall monthly cost of the table. This cost balance is indicative of a workload that regularly accesses or updates items already stored within DynamoDB.

When to select DynamoDB Standard-IA table class

DynamoDB Standard-IA table class is best suited for tables whose storage cost is approximately 50% or more of the overall monthly cost of the table. This cost balance is indicative of a workload that creates or reads fewer items per month than it keeps in storage.

A common use for the Standard-IA table class is moving less frequently accessed data to individual Standard-IA tables. For further information, see Optimizing the storage costs of your workloads with Amazon DynamoDB Standard-IA table class.

Additional factors to consider when choosing a table class

When deciding between the two table classes, there are some additional factors worth considering as part of your decision.

Reserved capacity

Purchasing reserved capacity for tables using the Standard-IA table class is currently not supported. When transitioning from a Standard table with reserved capacity to a Standard-IA table without reserved capacity, you may not see a cost benefit.

Identify your unused resources in DynamoDB

This section provides an overview of how to evaluate your unused resources regularly. As your application requirements evolve you should ensure no resources are unused and contributing to unnecessary Amazon DynamoDB costs. The procedures described below will use Amazon CloudWatch metrics to identify unused resources and will help you identify and take action on those resources to reduce costs.

You can monitor DynamoDB using CloudWatch, which collects and processes raw data from DynamoDB into readable, near real-time metrics. These statistics are retained for a period of time, so that you can access historical information to better understand your utilization. By default, DynamoDB metric data is sent to CloudWatch automatically. For more information, see What is Amazon CloudWatch? and Metrics retention in the *Amazon CloudWatch User Guide*.

Topics

- How to identify unused resources
- Identifying unused table resources
- Cleaning up unused table resources

- Identifying unused GSI resources
- Cleaning up unused GSI resources
- Cleaning up unused global tables
- Cleaning up unused backups or point-in-time recovery (PITR)

How to identify unused resources

To identify unused tables or indexes, we'll look at the following CloudWatch metrics over a period of 30 days to understand if there are any active reads or writes on the table or any reads on the global secondary indexes (GSIs):

ConsumedReadCapacityUnits

The number of read capacity units consumed over the specified time period, so you can track how much consumed capacity you have used. You can retrieve the total consumed read capacity for a table and all of its global secondary indexes, or for a particular global secondary index.

ConsumedWriteCapacityUnits

The number of write capacity units consumed over the specified time period, so you can track how much consumed capacity you have used. You can retrieve the total consumed write capacity for a table and all of its global secondary indexes, or for a particular global secondary index.

Identifying unused table resources

Amazon CloudWatch is a monitoring and observability service which provides the DynamoDB table metrics you'll use to identify unused resources. CloudWatch metrics can be viewed through the AWS Management Console as well as through the AWS Command Line Interface.

AWS Command Line Interface

To view your tables metrics through the AWS Command Line Interface, you can use the following commands.

1. First, evaluate your table's reads:

```
aws cloudwatch get-metric-statistics --metric-name
ConsumedReadCapacityUnits --start-time <start-time> --end-time <end-
time> --period <period> --namespace AWS/DynamoDB --statistics Sum --
```

```
dimensions Name=TableName,Value=<table-name>
```

To avoid falsely identifying a table as unused, evaluate metrics over a longer period. Choose an appropriate start-time and end-time range, such as **30 days**, and an appropriate period, such as **86400**.

In the returned data, any **Sum** above **0** indicates that the table you are evaluating received read traffic during that period.

The following result shows a table receiving read traffic in the evaluated period:

```
        {
            "Timestamp": "2022-08-25T19:40:00Z",
            "Sum": 36023355.0,
            "Unit": "Count"
        },
        {
            "Timestamp": "2022-08-12T19:40:00Z",
            "Sum": 38025777.5,
            "Unit": "Count"
        },
```

The following result shows a table not receiving read traffic in the evaluated period:

```
        {
            "Timestamp": "2022-08-01T19:50:00Z",
            "Sum": 0.0,
            "Unit": "Count"
        },
        {
            "Timestamp": "2022-08-20T19:50:00Z",
            "Sum": 0.0,
            "Unit": "Count"
        },
```

2. Next, evaluate your table's writes:

```
aws cloudwatch get-metric-statistics --metric-name
ConsumedWriteCapacityUnits --start-time <start-time> --end-time <end-
time> --period <period> --namespace AWS/DynamoDB --statistics Sum --
dimensions Name=TableName,Value=<table-name>
```

To avoid falsely identifying a table as unused, you will want to evaluate metrics over a longer period. Choose an appropriate start-time and end-time range, such as **30 days**, and an appropriate period, such as **86400**.

In the returned data, any **Sum** above **0** indicates that the table you are evaluating received read traffic during that period.

The following result shows a table receiving write traffic in the evaluated period:

```
{
    "Timestamp": "2022-08-19T20:15:00Z",
    "Sum": 41014457.0,
    "Unit": "Count"
},
{
    "Timestamp": "2022-08-18T20:15:00Z",
    "Sum": 40048531.0,
    "Unit": "Count"
},
```

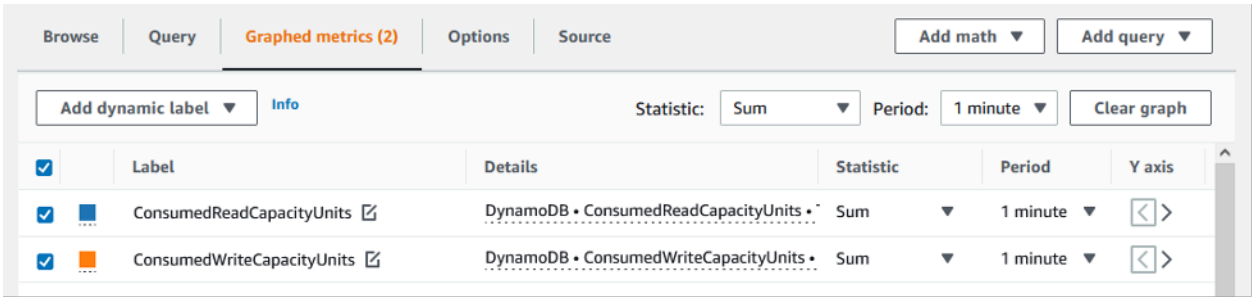
The following result shows a table not receiving write traffic in the evaluated period:

```
{
    "Timestamp": "2022-07-31T20:15:00Z",
    "Sum": 0.0,
    "Unit": "Count"
},
{
    "Timestamp": "2022-08-19T20:15:00Z",
    "Sum": 0.0,
    "Unit": "Count"
},
```

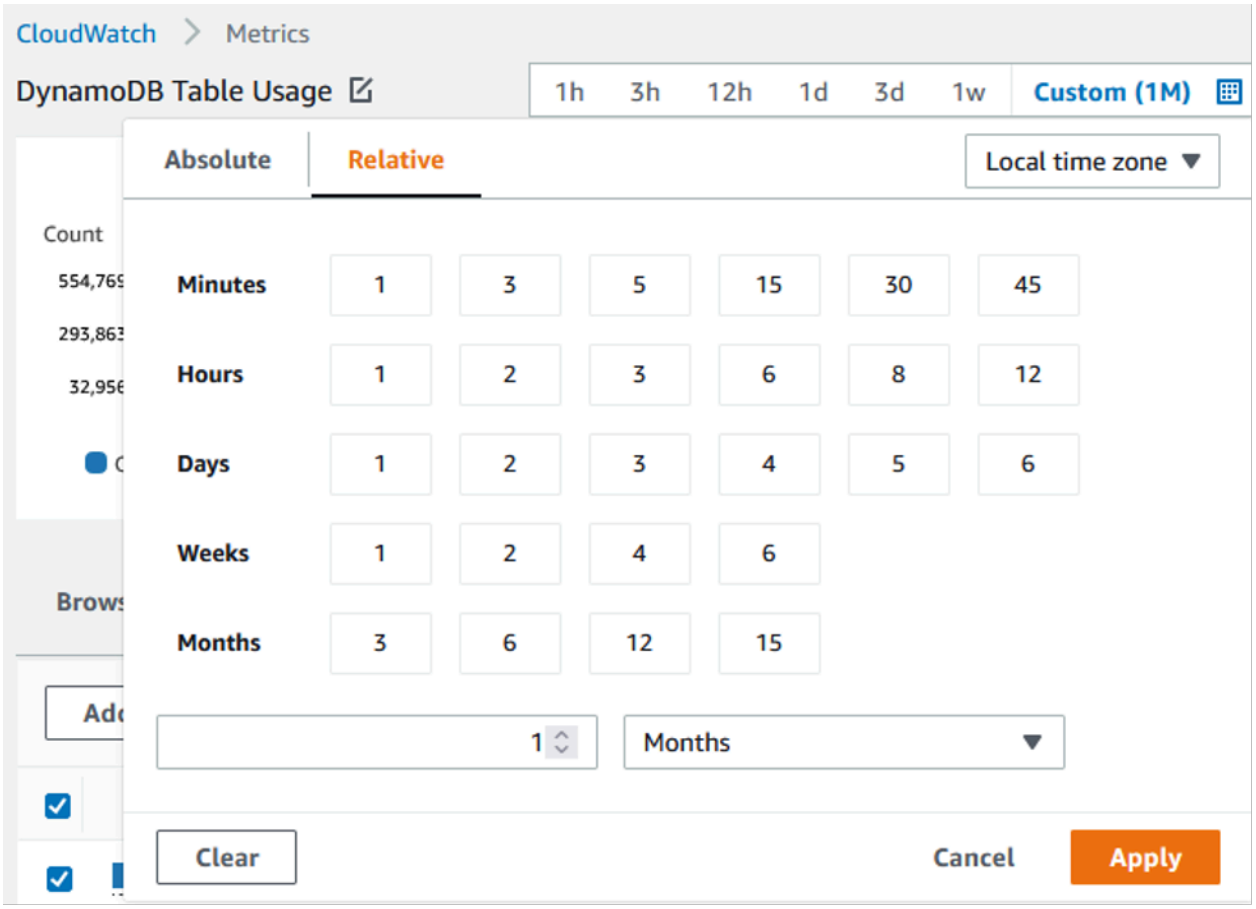
AWS Management Console

The following steps will allow you to evaluate your resources utilization through the AWS Management Console.

1. Log into the AWS console and navigate to the CloudWatch service page at https://console.aws.amazon.com/cloudwatch/. Select the appropriate AWS region in the top right of the console, if necessary.
2. On the left navigation bar, locate the Metrics section and select **All metrics**.
3. The action above will open a dashboard with two panels. In the top panel you will see currently graphed metrics. In the bottom you will select the metrics available to graph. Select DynamoDB in the bottom panel.
4. In the DynamoDB metrics selection panel select the **Table Metrics** category to show the metrics for your tables in the current region.
5. Identify your table name by scrolling down the menu, then select the metrics `ConsumedReadCapacityUnits` and `ConsumedWriteCapacityUnits` for your table.
6. Select the **Graphed metrics (2)** tab and adjust the **Statistic** column to **Sum**.

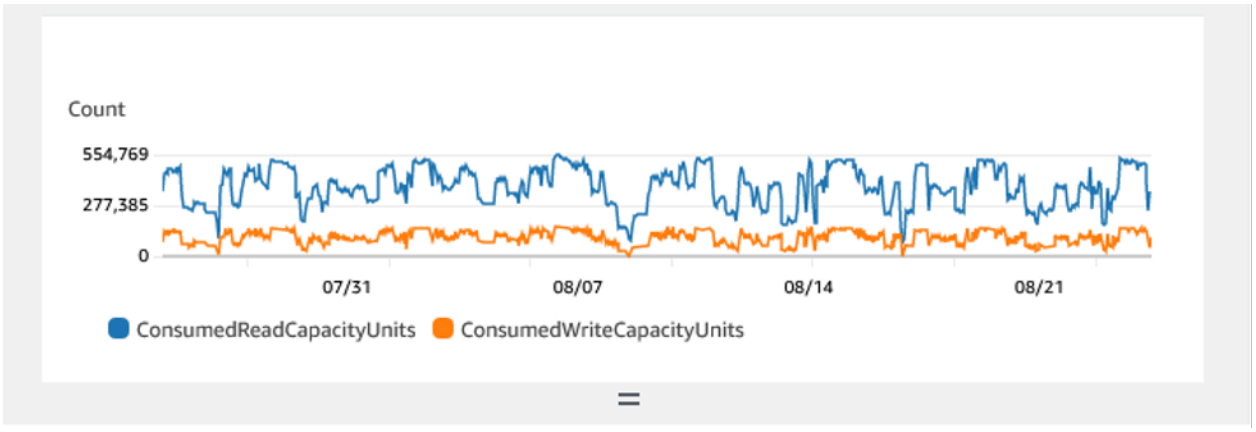


7. To avoid falsely identifying a table as unused, you'll want to evaluate metrics over a longer period. At the top of the graph panel choose an appropriate time frame, such as 1 month, to evaluate your table. Select **Custom**, select **1 Months** in the dropdowns, and choose **Apply**.

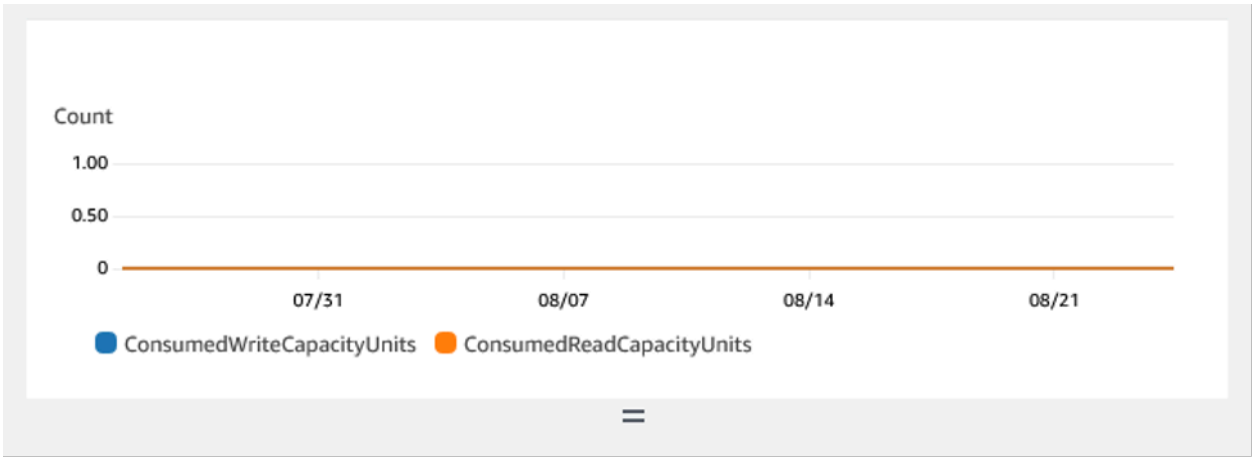


8. Evaluate the graphed metrics for your table to determine if it is being used. Metrics that have gone above **0** indicate that a table has been used during the evaluated time period. A flat graph at **0** for both read and write indicates a table that is unused.

   The following image shows a table with read traffic:



   The following image shows a table without read traffic:



Cleaning up unused table resources

If you have identified unused table resources, you can reduce their ongoing costs in the following ways.

> **Note**
>
> If you have identified an unused table but would still like to keep it available in case it needs to be accessed in the future, consider switching it to on-demand mode. Otherwise, you can consider backing up and deleting the table entirely.

Capacity modes

DynamoDB charges for reading, writing, and storing data in your DynamoDB tables.

DynamoDB has two capacity modes, which come with specific billing options for processing reads and writes on your tables: on-demand and provisioned. The read/write capacity mode controls how you are charged for read and write throughput and how you manage capacity.

For on-demand mode tables, you don't need to specify how much read and write throughput you expect your application to perform. DynamoDB charges you for the reads and writes that your application performs on your tables in terms of read request units and write request units. If there is no activity on your table/index you do not pay for throughput but you'll still incur a storage charge.

Table class

DynamoDB also offers two table classes designed to help you optimize for cost. The DynamoDB Standard table class is the default and is recommended for most workloads. The DynamoDB Standard-Infrequent Access (DynamoDB Standard-IA) table class is optimized for tables where storage is the dominant cost.

If there is no activity on your table or index, storage is likely to be the dominant cost and changing table class will offer a significant savings.

Deleting tables

If you've discovered an unused table and would like to delete it, you may wish to make a backup or export of the data first.

Backups taken through AWS Backup can leverage cold storage tiering, further reducing costs. Refer to the Using AWS Backup with DynamoDB documentation for information on how enable backups through AWS Backup as well as the Managing backup plans documentation for information on how to use lifecycle to move your backup to cold storage.

Alternatively, you may choose to export your table's data to S3. To do so, refer to the Export to Amazon S3 documentation. Once your data is exported, if you wish to leverage S3 Glacier Instant Retrieval, S3 Glacier Flexile Retrieval, or S3 Glacier Deep Archive to further reduce costs, see Managing your storage lifecycle.

After your table has been backed up, you may choose to delete it either through the AWS Management Console or through the AWS Command Line Interface.

Identifying unused GSI resources

The steps for identifying an unused global secondary are similar to those for identifying an unused table. Since DynamoDB replicates items written to your base table into your GSI if they contain the attribute used as the GSI's partition key, an unused GSI is still likely to have `ConsumedWriteCapacityUnits` above 0 if its base table is in use. As a result, you'll be evaluating only the `ConsumedReadCapacityUnits` metric to determine if your GSI is unused.

To view your GSI metrics through the AWS AWS CLI, you can use the following commands to evaluate your tables reads:

```
aws cloudwatch get-metric-statistics --metric-name
ConsumedReadCapacityUnits --start-time <start-time> --end-time <end-
time> --period <period> --namespace AWS/DynamoDB --statistics Sum --
dimensions Name=TableName,Value=<table-name>
```

```
Name=GlobalSecondaryIndexName,Value=<index-name>
```

To avoid falsely identifying a table as unused, you will want to evaluate metrics over a longer period. Choose an appropriate start-time and end-time range, such as 30 days, and an appropriate period, such as 86400.

In the returned data, any Sum above 0 indicates that the table you are evaluating received read traffic during that period.

The following result shows a GSI receiving read traffic in the evaluated period:

```
        {
          "Timestamp": "2022-08-17T21:20:00Z",
          "Sum": 36319167.0,
          "Unit": "Count"
        },
        {
          "Timestamp": "2022-08-11T21:20:00Z",
          "Sum": 1869136.0,
          "Unit": "Count"
        },
```

The following result shows a GSI receiving minimal read traffic in the evaluated period:

```
        {
          "Timestamp": "2022-08-28T21:20:00Z",
          "Sum": 0.0,
          "Unit": "Count"
        },
        {
          "Timestamp": "2022-08-15T21:20:00Z",
          "Sum": 2.0,
          "Unit": "Count"
        },
```

The following result shows a GSI receiving no read traffic in the evaluated period:

```
        {
          "Timestamp": "2022-08-17T21:20:00Z",
          "Sum": 0.0,
          "Unit": "Count"
        },
```

```
        {
          "Timestamp": "2022-08-11T21:20:00Z",
          "Sum": 0.0,
          "Unit": "Count"
        },
```

Cleaning up unused GSI resources

If you've identified an unused GSI, you can choose to delete it. Since all data present in a GSI is also present in the base table, additional backup is not necessary before deleting a GSI. If in the future the GSI is once again needed, it may be added back to the table.

If you have identified an infrequently used GSI, you should consider design changes in your application that would allow you to delete it or reduce its cost. For example, while DynamoDB scans should be used sparingly because they can consume large amounts of system resources, they may be more cost effective than a GSI if the access pattern it supports is used very infrequently.

Additionally, if a GSI is required to support an infrequent access pattern consider projecting a more limited set of attributes. While this may require subsequent queries against the base table to support your infrequent access patterns, it can potentially offer a significant reduction in storage and write costs.

Cleaning up unused global tables

Amazon DynamoDB global tables provide a fully managed solution for deploying a multi-Region, multi-active database, without having to build and maintain your own replication solution.

Global tables are ideal for providing low-latency access to data close to users and as well as a secondary region for disaster recovery.

If the global tables option is enabled for a resource in an effort to provide low-latency access to data but is not part of your disaster recovery strategy, validate that both replicas are actively serving read traffic by evaluating their CloudWatch metrics. If one replica does not serve read traffic, it may be an unused resource.

If global tables are part of your disaster recovery strategy, one replica not receiving read traffic may be expected under an active/standby pattern.

Cleaning up unused backups or point-in-time recovery (PITR)

DynamoDB offers two styles of backup. Point-in-time recovery provides continuous backups for up to 35 days to help you protect against accidental writes or deletes while on-demand backup allows

for snapshot creation which can be saved long term. You can set the recovery period to any value between 1 and 35 days. Both types of backups have costs associated with them.

Refer to the documentation for Backup and restore for DynamoDB and Point-in-time backups for DynamoDB to determine if your tables have backups enabled that may no longer be needed.

Evaluate your DynamoDB table usage patterns

This section provides an overview of how to evaluate if you are efficiently using your DynamoDB tables. There are certain usage patterns which are not optimal for DynamoDB, and they allow room for optimization from both a performance and cost perspective.

Topics

- Perform fewer strongly-consistent read operations
- Perform fewer transactions for read operations
- Perform fewer scans
- Shorten attribute names
- Enable Time to Live (TTL)
- Replace global tables with cross-Region backups

Perform fewer strongly-consistent read operations

DynamoDB allows you to configure read consistency on a per-request basis. Read requests are eventually consistent by default. Eventually consistent reads are charged at 0.5 RCU for upto 4 KB of data.

Most parts of distributed workloads are flexible and can tolerate eventual consistency. However, there can be access patterns requiring strongly consistent reads. Strongly consistent reads are charged at 1 RCU for upto 4 KB of data, essentially doubling your read costs. DynamoDB provides you with the flexibility to use both consistency models on the same table.

You can evaluate your workload and application code to confirm if strongly consistent reads are used only where required.

Perform fewer transactions for read operations

DynamoDB allows you to group certain actions in an all-or-nothing manner, which means you have the ability to execute ACID transactions with DynamoDB. However, as is the case with relational databases, it is not necessary to follow this approach for every action.

A transactional read operation of up to 4 KB consumes 2 RCUs as opposed to the default 0.5 RCUs for reading the same amount of data in an eventually consistent manner. The costs are doubled in write operations which means, a transactional write of up to 1 KB equates to 2 WCUs.

To determine if all operations on your tables are transactions, CloudWatch metrics for the table can be filtered down to the transaction APIs. If transaction APIs are the only graphs available under the `SuccessfulRequestLatency` metrics for the table, this would confirm that every operation is a transaction for this table. Alternatively, if the overall capacity utilization trend matches the transaction API trend, consider revisiting the application design as it seems dominated by transactional APIs.

Perform fewer scans

The extensive use of `Scan` operations generally stems from the need to run analytical queries on the DynamoDB data. Running frequent `Scan` operations on large table can be inefficient and expensive.

A better alternative is to use the Export to S3 feature and choosing a point in time to export the table state to S3. AWS offers services like Athena which can then be used to run analytical queries on the data without consuming any capacity from the table.

The frequency for `Scan` operations can be determined using the `SampleCount` statistic under the `SuccessfulRequestLatency` metric for `Scan`. If `Scan` operations are indeed very frequent, the access patterns and data model should be re-evaluated.

Shorten attribute names

The total size of an item in DynamoDB is the sum of its attribute name lengths and values. Having long attribute names not only contributes towards storage costs, but it might also lead to higher RCU/WCU consumption. We recommend that you choose shorter attribute names rather than long ones. Having shorter attribute names can help limit the item size within the next 4KB/1KB boundary after which you would consume additional RCU/WCU to access data.

Enable Time to Live (TTL)

Time to Live (TTL) can identify items older than the expiry time that you have set on an item and remove them from the table. If your data grows over time and older data becomes irrelevant, enabling TTL on the table can help trim your data down and save on storage costs.

Another useful aspect of TTL is that the expired items occur on your DynamoDB streams, so rather than just removing the data from your data, it is possible to consume those items from the stream

and archive them to a lower cost storage tier. Additionally, deleting items via TTL comes at no additional cost — it does not consume capacity, and there's no overhead of designing a clean up application.

Replace global tables with cross-Region backups

Global tables allow you to maintain multiple active replica tables in different Regions — they can all accept write operations and replicate data across each other. It is easy to set up replicas and the synchronization is managed for you. The replicas converge to a consistent state using a last writer wins strategy.

If you are using Global tables purely as a part of failover or disaster recovery (DR) strategy, you can replace them with a cross-Region backup copy for relatively lenient recovery point objectives and recovery time objective requirements. If you do not require fast local access and the high availability of five nines, maintaining a global table replica might not be the best approach for failover.

As an alternative, consider using AWS Backup to manage DynamoDB backups. You can schedule regular backups and copy them across Regions to meet DR requirements in a more cost-effective approach compared to using Global tables.

Evaluate your DynamoDB streams usage

This section provides an overview of how to evaluate your DynamoDB Streams usage. There are certain usage patterns which are not optimal for DynamoDB, and they allow room for optimization from both a performance and cost perspective.

You have two native streaming integrations for streaming and event-driven use cases:

- Amazon DynamoDB Streams
- Amazon Kinesis Data Streams

This page will just focus on cost optimization strategies for these options. If you'd like to instead find out how to choose between the two options, see Streaming options for change data capture.

Topics

- Optimizing costs for DynamoDB Streams
- Optimizing costs for Kinesis Data Streams

- Cost optimization strategies for both types of Streams usage

Optimizing costs for DynamoDB Streams

As mentioned in the pricing page for DynamoDB Streams, regardless of the table's throughput capacity mode, DynamoDB charges on the number of read requests made towards the table's DynamoDB Stream. Read requests made towards a DynamoDB Stream are different from the read requests made towards a DynamoDB table.

Each read request in terms of the stream is in the form of a `GetRecords` API call that can return up to 1000 records or 1 MB worth of records in the response, whichever is reached first. None of the other DynamoDB Stream APIs are charged and DynamoDB Streams are not charged for being idle. In other words, if no read requests are made to a DynamoDB Stream, no charges will be incurred for having a DynamoDB Stream enabled on a table.

Here are a few consumer applications for DynamoDB Streams:

- AWS Lambda function(s)
- Amazon Kinesis Data Streams-based applications
- Customer consumer applications built using an AWS SDK

Read requests made by AWS Lambda-based consumers of DynamoDB Streams are free, whereas calls made by consumers of any other kind are charged. Every month, the first 2,500,000 read requests made by non-Lambda consumers are also free of cost. This applies to all read requests made to any DynamoDB Streams in an AWS Account for each AWS Region.

Monitoring your DynamoDB Streams usage

DynamoDB Streams charges on the billing console are grouped together for all DynamoDB Streams across the AWS Region in an AWS Account. Currently, tagging DynamoDB Streams is not supported, so cost allocation tags cannot be used to identify granular costs for DynamoDB Streams. The volume of `GetRecords` calls can be obtained at the DynamoDB Stream level to compute the charges per stream. The volume is represented by the DynamoDB Stream's CloudWatch metric `SuccessfulRequestLatency` and its `SampleCount` statistic. This metric will also include `GetRecords` calls made by global tables to perform on-going replication as well as calls made by AWS Lambda consumers, both of which are not charged. For information on other CloudWatch metrics published by DynamoDB Streams, see DynamoDB Metrics and dimensions.

Using AWS Lambda as the consumer

Evaluate if using AWS Lambda functions as the consumers for DynamoDB Streams is feasible because that can eliminate costs associated with reading from the DynamoDB Stream. On the other hand, DynamoDB Streams Kinesis Adapter or SDK based consumer applications will be charged on the number of `GetRecords` calls they make towards the DynamoDB Stream.

Lambda function invocations will be charged based on standard Lambda pricing, however no charges will be incurred by DynamoDB Streams. Lambda will poll shards in your DynamoDB Stream for records at a base rate of 4 times per second. When records are available, Lambda invokes your function and waits for the result. If processing succeeds, Lambda resumes polling until it receives more records.

Tuning DynamoDB Streams Kinesis Adapter-based consumer applications

Since read requests made by non-Lambda based consumers are charged for DynamoDB Streams, it is important to find a balance between the near real-time requirement and the number of times the consumer application must poll the DynamoDB Stream.

The frequency of polling DynamoDB Streams using a DynamoDB Streams Kinesis Adapter based application is determined by the configured `idleTimeBetweenReadsInMillis` value. This parameter determines the amount of time in milliseconds that the consumer should wait before processing a shard in case the previous `GetRecords` call made to the same shard did not return any records. By default, this value for this parameter is 1000 ms. If near real-time processing is not required, this parameter could be increased to have the consumer application make fewer `GetRecords` calls and optimize on DynamoDB Streams calls.

Optimizing costs for Kinesis Data Streams

When a Kinesis Data Stream is set as the destination to deliver change data capture events for a DynamoDB table, the Kinesis Data Stream may need separate sizing management which will affect the overall costs. DynamoDB charges in terms of Change Data capture Units (CDUs) where each unit is a made of up a 1 KB DynamoDB item size attempted by the DynamoDB service to the destination Kinesis Data Stream.

In addition to charges by the DynamoDB service, standard Kinesis Data Stream charges will be incurred. As mentioned in the pricing page, the service pricing differs based on the capacity mode - provisioned and on-demand, which are distinct from DynamoDB table capacity modes and are user-defined. At a high level, Kinesis Data Streams charges an hourly rate based on the capacity mode, as well as on data ingested into the stream by DynamoDB service. There may be additional

charges like data retrieval (for on-demand mode), extended data retention (beyond default 24 hours), and enhanced fan-out consumer retrievals depending on the user configuration for the Kinesis Data Stream.

Monitoring your Kinesis Data Streams usage

Kinesis Data Streams for DynamoDB publishes metrics from DynamoDB in addition to standard Kinesis Data Stream CloudWatch Metrics. It may be possible that a `Put` attempt by the DynamoDB service is throttled by the Kinesis service because of insufficient Kinesis Data Streams capacity, or by dependent components like a AWS KMS service that may be configured to encrypt the Kinesis Data Stream data at rest.

To learn more about CloudWatch metrics published by DynamoDB service for the Kinesis Data Stream, see Monitoring change data capture with Kinesis Data Streams. In order to avoid additional costs of service retries due to throttles, it is important to right size the Kinesis Data Stream in case of Provisioned Mode.

Choosing the right capacity mode for Kinesis Data Streams

Kinesis Data Streams are supported in two capacity modes – provisioned mode and on-demand mode.

- If the workload involving Kinesis Data Stream has predictable application traffic, traffic that is consistent or ramps gradually, or traffic that can be forecasted accurately, then Kinesis Data Streams' **provisioned mode** is suitable and will be more cost efficient
- If the workload is new, has unpredictable application traffic, or you prefer not to manage capacity, then Kinesis Data Streams' **on-demand mode** is suitable and will be more cost efficient

A best practice to optimize costs would be to evaluate if the DynamoDB table associated with the Kinesis Data Stream has a predictable traffic pattern that can leverage provisioned mode of Kinesis Data Streams. If the workload is new, you could use on-demand mode for the Kinesis Data Streams for a few initial weeks, review the CloudWatch metrics to understand traffic patterns, and then switch the same Stream to provisioned mode based on the nature of the workload. In the case of provisioned mode, estimation on number shards can be made by following shard management considerations for Kinesis Data Streams.

Evaluate your consumer applications using Kinesis Data Streams for DynamoDB

Since Kinesis Data Streams don't charge on the number of `GetRecords` calls like DynamoDB Streams, consumer applications could make as many number of calls as possible, provided the

frequency is under the throttling limits for `GetRecords`. In terms of on-demand mode for Kinesis Data Streams, data reads are charged on a per GB basis. For provisioned mode Kinesis Data Streams, reads are not charged if the data is less than 7 days old. In the case of Lambda functions as Kinesis Data Streams consumers, Lambda polls each shard in your Kinesis Stream for records at a base rate of once per second.

Cost optimization strategies for both types of Streams usage

Event filtering for AWS Lambda consumers

Lambda event filtering allows you to discard events based on a filter criteria from being available in the Lambda function invocation batch. This optimizes Lambda costs for processing or discarding unwanted stream records within the consumer function logic. To learn more about configuring event filtering and writing your filtering criteria, see Lambda event filtering.

Tuning AWS Lambda consumers

Costs could be further be optimized by tuning Lambda configuration parameters like increasing the `BatchSize` to process more per invocation, enabling `BisectBatchOnFunctionError` to prevent processing duplicates (which incurs additional costs), and setting `MaximumRetryAttempts` to not run into too many retries. By default, failed consumer Lambda invocations are retried infinitely until the record expires from the stream, which is around 24 hours for DynamoDB Streams and configurable from 24 hours to up to 1 year for Kinesis Data Streams. Additional Lambda configuration options available including the ones mentioned above for DynamoDB Stream consumers are in the AWS Lambda developer guide.

Evaluate your provisioned capacity for right-sized provisioning in your DynamoDB table

This section provides an overview of how to evaluate if you have right-sized provisioning on your DynamoDB tables. As your workload evolves, you should modify your operational procedures appropriately, especially when your DynamoDB table is configured in provisioned mode and you have the risk to over-provision or under-provision your tables.

The procedures described below require statistical information that should be captured from the DynamoDB tables that are supporting your production application. To understand your application behavior, you should define a period of time that is significant enough to capture the data seasonality from your application. For example, if your application shows weekly patterns, using a three week period should give you enough room for analysing application throughput needs.

If you don't know where to start, use at least one month's worth of data usage for the calculations below.

While evaluating capacity, DynamoDB tables can configure **Read Capacity Units (RCUs)** and **Write Capacity Units (WCU)** independently. If your tables have any Global Secondary Indexes (GSI) configured, you will need to specify the throughput that it will consume, which will be also independent from the RCUs and WCUs from the base table.

> **Note**
>
> Local Secondary Indexes (LSI) consume capacity from the base table.

Topics

- How to retrieve consumption metrics on your DynamoDB tables
- How to identify under-provisioned DynamoDB tables
- How to identify over-provisioned DynamoDB tables

How to retrieve consumption metrics on your DynamoDB tables

To evaluate the table and GSI capacity, monitor the following CloudWatch metrics and select the appropriate dimension to retrieve either table or GSI information:

Read Capacity Units	Write Capacity Units
`ConsumedReadCapacityUnits`	`ConsumedWriteCapacityUnits`
`ProvisionedReadCapacityUnits`	`ProvisionedWriteCapacityUnits`
`ReadThrottleEvents`	`WriteThrottleEvents`

You can do this either through the AWS CLI or the AWS Management Console.

AWS CLI

Before we retrieve the table consumption metrics, we'll need to start by capturing some historical data points using the CloudWatch API.

Start by creating two files: `write-calc.json` and `read-calc.json`. These files will represent the calculations for a table or GSI. You'll need to update some of the fields, as indicated in the table below, to match your environment.

Field Name	Definition	Example
`<table-name>`	The name of the table that you will be analysing	SampleTable
`<period>`	The period of time that you will be using to evaluate the utilization target, based in seconds	For a 1-hour period you should specify: 3600
`<start-time>`	The beginning of your evaluation interval, specified in ISO8601 format	2022-02-21T23:00:00
`<end-time>`	The end of your evaluatio n interval, specified in ISO8601 format	2022-02-22T06:00:00

The write calculations file will retrieve the number of WCU provisioned and consumed in the time period for the date range specified. It will also generate a utilization percentage that will be used for analysis. The full content of the `write-calc.json` file should look like this:

```
{
  "MetricDataQueries": [
    {
      "Id": "provisionedWCU",
      "MetricStat": {
        "Metric": {
          "Namespace": "AWS/DynamoDB",
          "MetricName": "ProvisionedWriteCapacityUnits",
          "Dimensions": [
            {
              "Name": "TableName",
              "Value": "<table-name>"
            }
```

```
          ]
        },
        "Period": <period>,
        "Stat": "Average"
      },
      "Label": "Provisioned",
      "ReturnData": false
    },
    {
      "Id": "consumedWCU",
      "MetricStat": {
        "Metric": {
          "Namespace": "AWS/DynamoDB",
          "MetricName": "ConsumedWriteCapacityUnits",
          "Dimensions": [
            {
              "Name": "TableName",
              "Value": "<table-name>""
            }
          ]
        },
        "Period": <period>,
        "Stat": "Sum"
      },
      "Label": "",
      "ReturnData": false
    },
    {
      "Id": "m1",
      "Expression": "consumedWCU/PERIOD(consumedWCU)",
      "Label": "Consumed WCUs",
      "ReturnData": false
    },
    {
      "Id": "utilizationPercentage",
      "Expression": "100*(m1/provisionedWCU)",
      "Label": "Utilization Percentage",
      "ReturnData": true
    }
  ],
  "StartTime": "<start-time>",
  "EndTime": "<ent-time>",
  "ScanBy": "TimestampDescending",
  "MaxDatapoints": 24
```

```
}
```

The read calculations file uses a similar file. This file will retrieve how many RCUs were provisioned and consumed during the time period for the date range specified. The contents of the `read-calc.json` file should look like this:

```
{
  "MetricDataQueries": [
    {
      "Id": "provisionedRCU",
      "MetricStat": {
        "Metric": {
          "Namespace": "AWS/DynamoDB",
          "MetricName": "ProvisionedReadCapacityUnits",
          "Dimensions": [
            {
              "Name": "TableName",
              "Value": "<table-name>"
            }
          ]
        },
        "Period": <period>,
        "Stat": "Average"
      },
      "Label": "Provisioned",
      "ReturnData": false
    },
    {
      "Id": "consumedRCU",
      "MetricStat": {
        "Metric": {
          "Namespace": "AWS/DynamoDB",
          "MetricName": "ConsumedReadCapacityUnits",
          "Dimensions": [
            {
              "Name": "TableName",
              "Value": "<table-name>"
            }
          ]
        },
        "Period": <period>,
        "Stat": "Sum"
      },
```

```
      "Label": "",
      "ReturnData": false
    },
    {
      "Id": "m1",
      "Expression": "consumedRCU/PERIOD(consumedRCU)",
      "Label": "Consumed RCUs",
      "ReturnData": false
    },
    {
      "Id": "utilizationPercentage",
      "Expression": "100*(m1/provisionedRCU)",
      "Label": "Utilization Percentage",
      "ReturnData": true
    }
  ],
  "StartTime": "<start-time>",
  "EndTime": "<end-time>",
  "ScanBy": "TimestampDescending",
  "MaxDatapoints": 24
}
```

One you've created the files, you can start retrieving utilization data.

1. To retreive the write utilization data, issue the following command:

   ```
   aws cloudwatch get-metric-data --cli-input-json file://write-calc.json
   ```

2. To retreive the read utilization data, issue the following command:

   ```
   aws cloudwatch get-metric-data --cli-input-json file://read-calc.json
   ```

The result for both queries will be a series of data points in JSON format that will be used for analysis. Your result will depend on the number of data points you specified, the period, and your own specific workload data. It could look something like this:

```
{
    "MetricDataResults": [
        {
            "Id": "utilizationPercentage",
            "Label": "Utilization Percentage",
```

```
            "Timestamps": [
                "2022-02-22T05:00:00+00:00",
                "2022-02-22T04:00:00+00:00",
                "2022-02-22T03:00:00+00:00",
                "2022-02-22T02:00:00+00:00",
                "2022-02-22T01:00:00+00:00",
                "2022-02-22T00:00:00+00:00",
                "2022-02-21T23:00:00+00:00"
            ],
            "Values": [
                91.55364583333333,
                55.066631944444445,
                2.6114930555555556,
                24.9496875,
                40.94725694444445,
                25.61819444444444,
                0.0
            ],
            "StatusCode": "Complete"
        }
    ],
    "Messages": []
}
```

> **Note**
>
> If you specify a short period and a long time range, you might need to modify the `MaxDatapoints` which is by default set to 24 in the script. This represents one data point per hour and 24 per day.

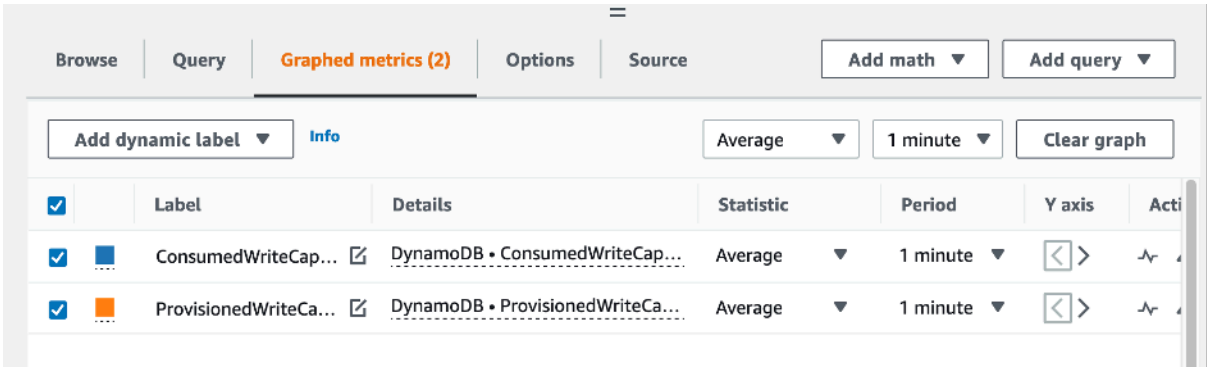
AWS Management Console

1. Log into the AWS Management Console and navigate to the CloudWatch service page. Select an appropriate AWS Region if necessary.
2. Locate the **Metrics** section on the left navigation bar and select **All metrics**.
3. This will open a dashboard with two panels. The top panel will show you the graphic, and the bottom panel will show the metrics you want to graph. Choose **DynamoDB**.
4. Choose **Table Metrics**. This will show you the tables in your current Region.

5. Use the Search box to search for your table name and choose the write operation metrics: `ConsumedWriteCapacityUnits` and `ProvisionedWriteCapacityUnits`

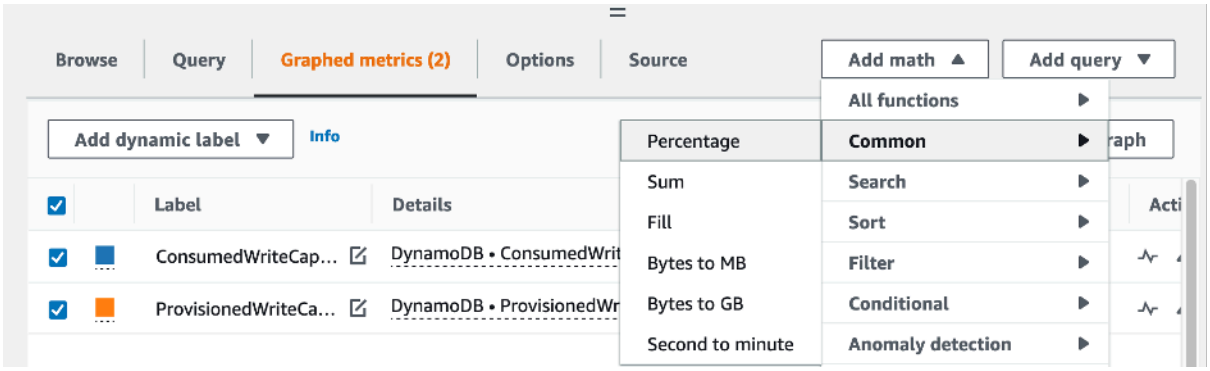
> **Note**
>
> This example talks about write operation metrics, but you can also use these steps to graph the read operation metrics.

6. Choose the **Graphed metrics (2)** tab to modify the formulas. By default, CloudWatch selects the statistical function **Average** for the graphs.

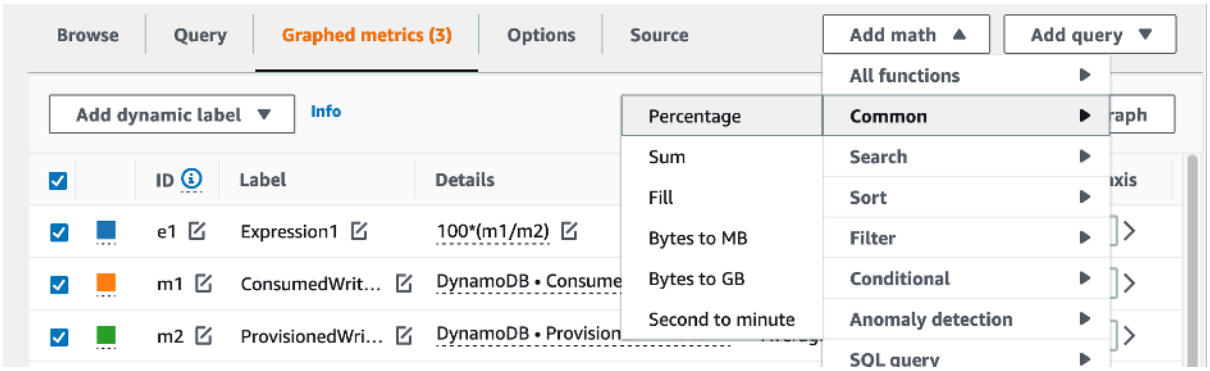


7. While having both graphed metrics selected (the checkbox on the left) select the menu **Add math**, followed by **Common**, and then select the **Percentage** function. Repeat the procedure twice.

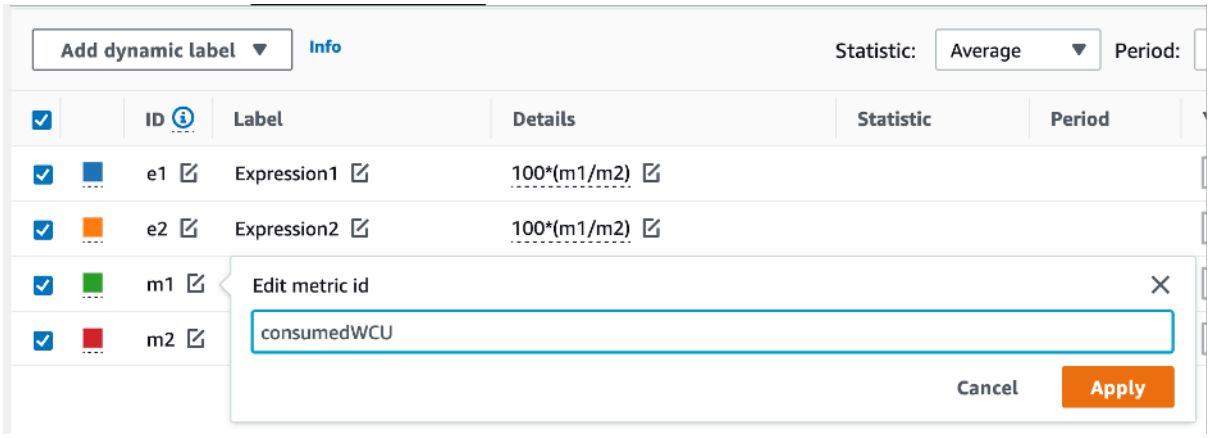
   First time selecting the **Percentage** function:



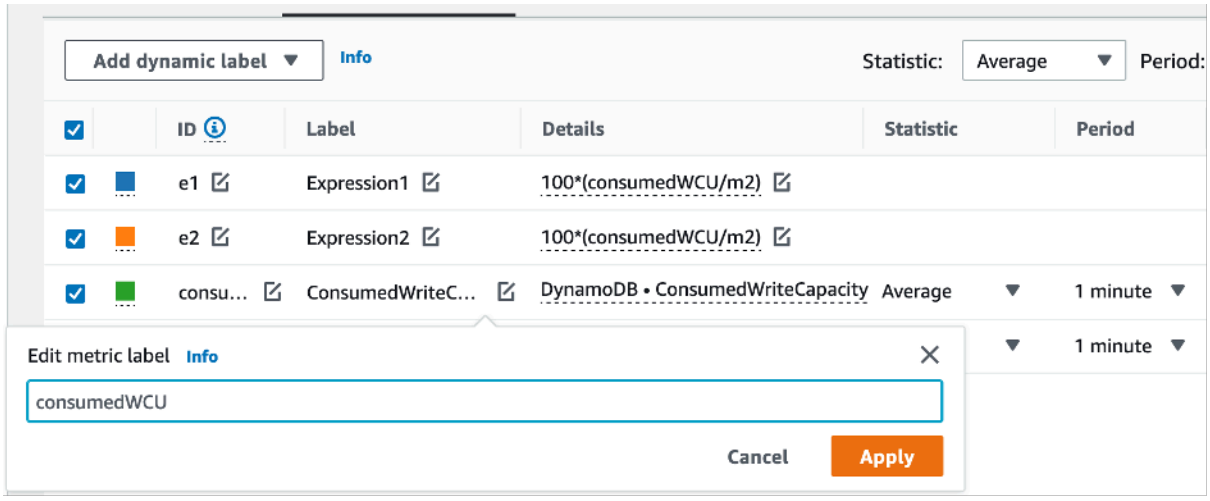
   Second time selecting the **Percentage** function:



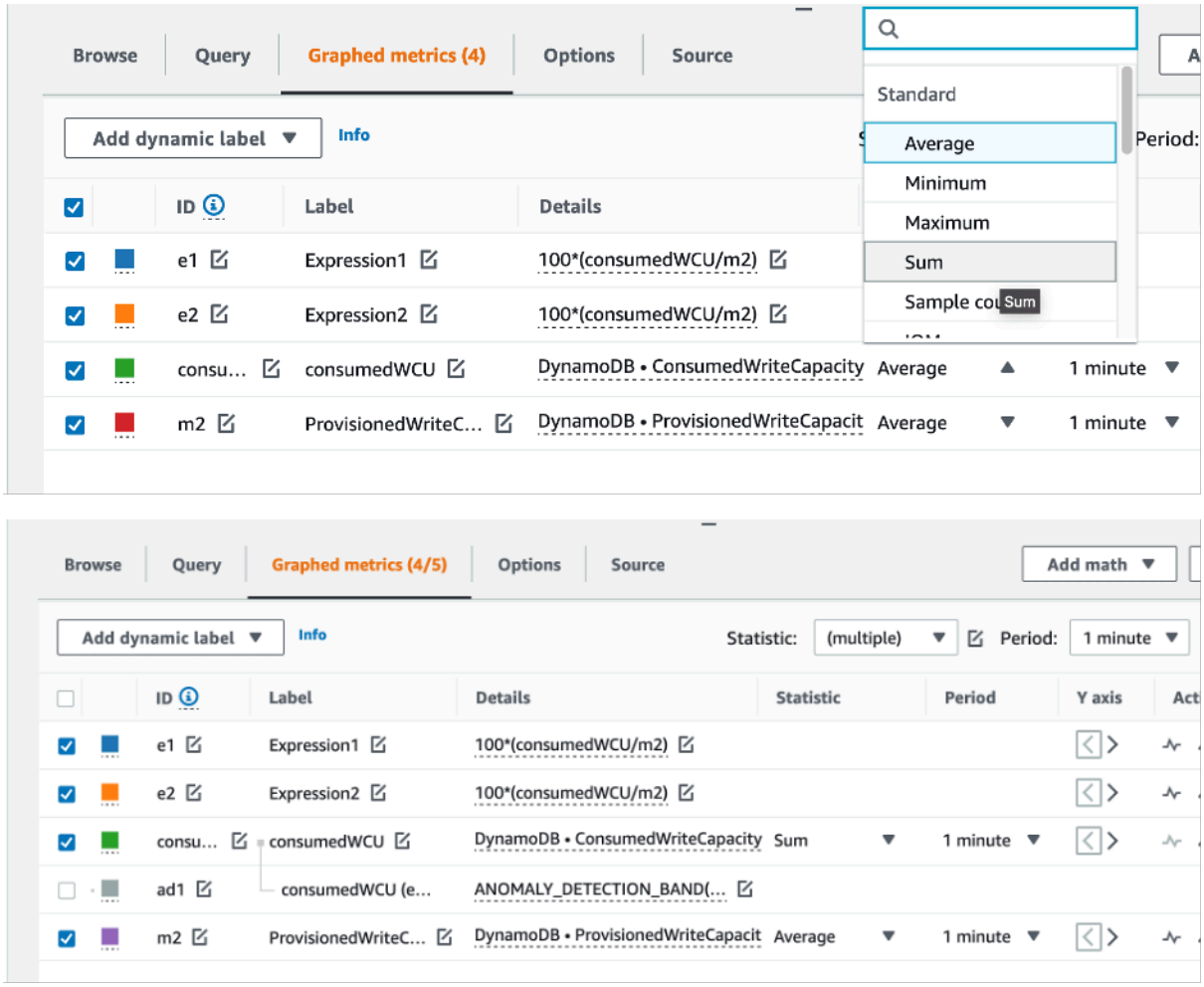
8. At this point you should have four metrics in the bottom menu. Let's work on the `ConsumedWriteCapacityUnits` calculation. To be consistent, we need to match the names for the ones we used in the AWS CLI section. Click on the **m1 ID** and change this value to **consumedWCU**.



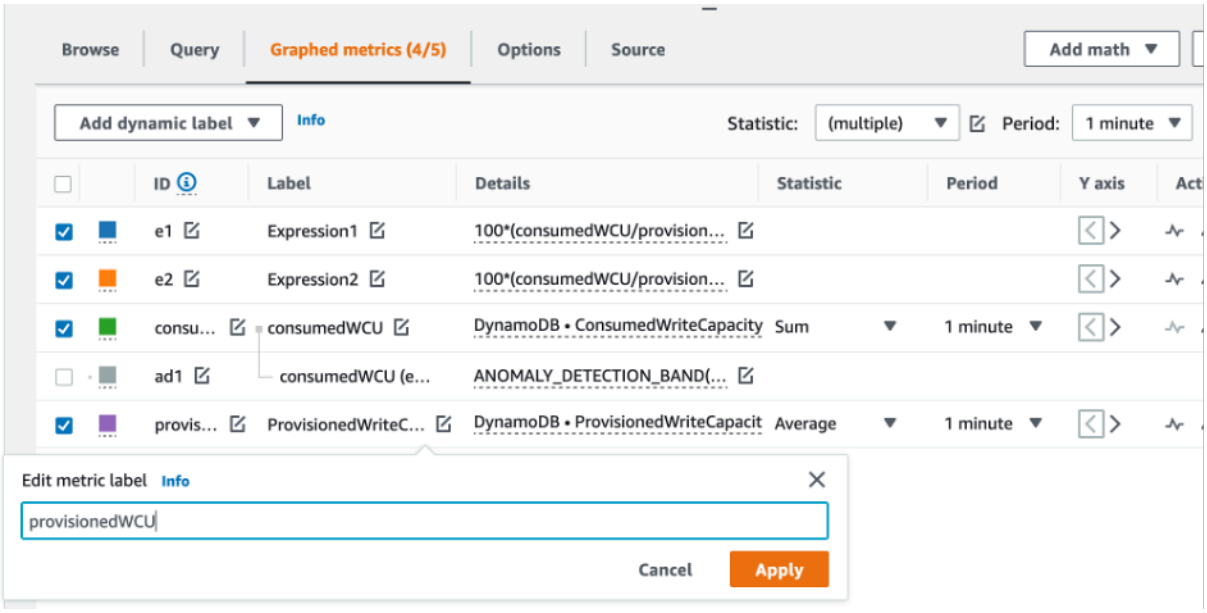
Rename the **ConsumedWriteCapacityUnit** label as **consumedWCU**.



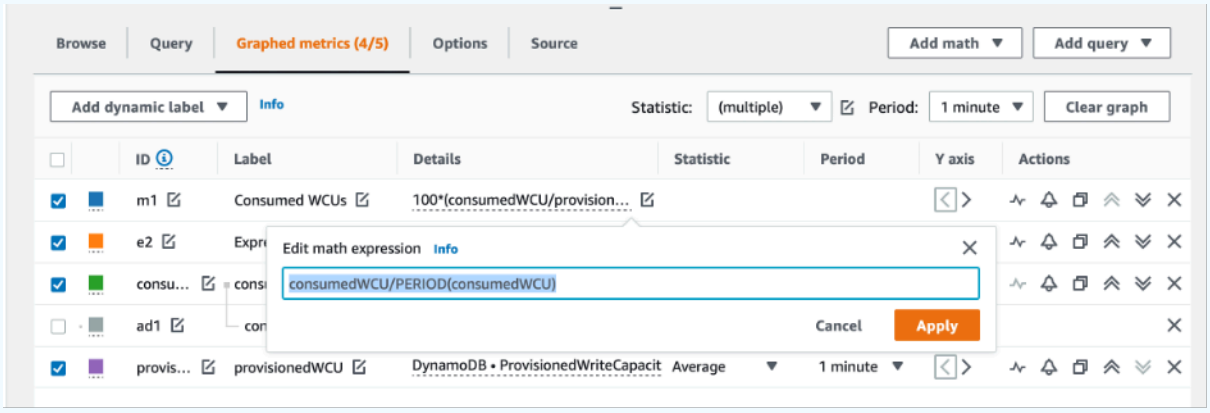
9. Change the statistic from **Average** to **Sum**. This action will automatically create another metric called **ANOMALY_DETECTION_BAND**. For the scope of this procedure, let's ignore it by removing the checkbox on the newly generated **ad1 metric**.



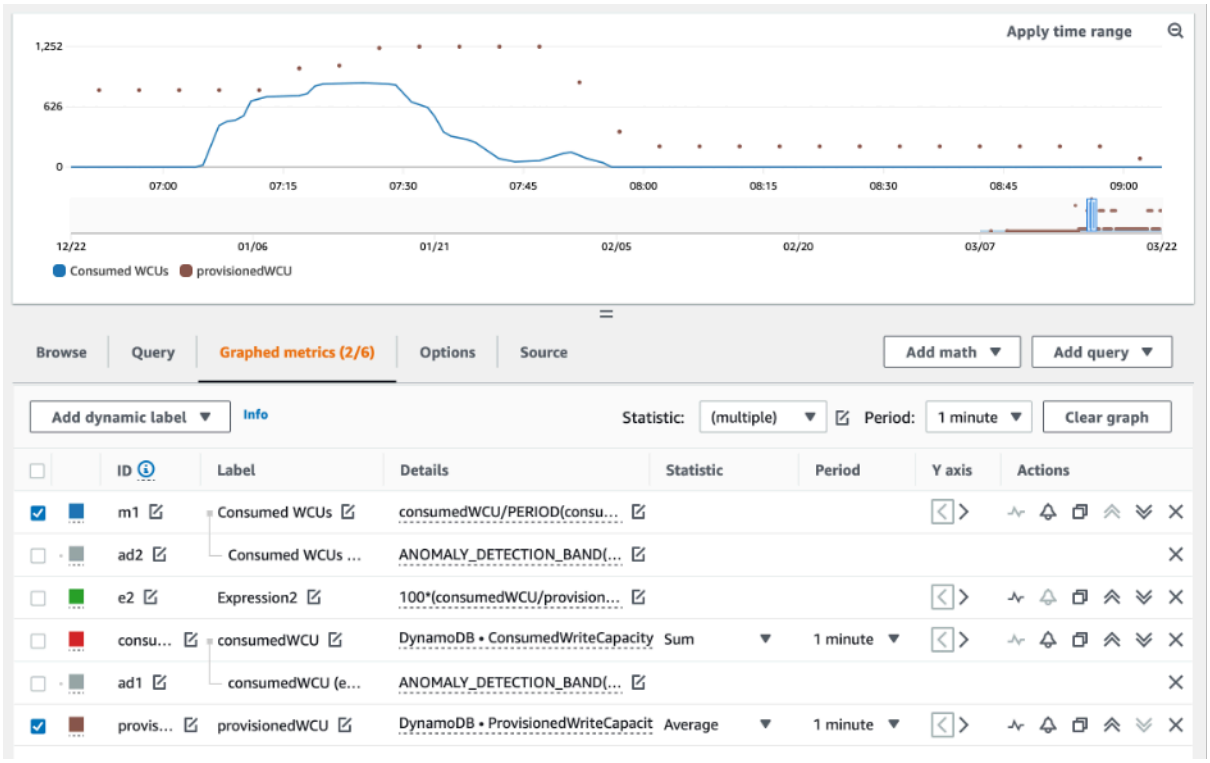
10. Repeat step 8 to rename the **m2 ID** to **provisionedWCU**. Leave the statistic set to **Average**.



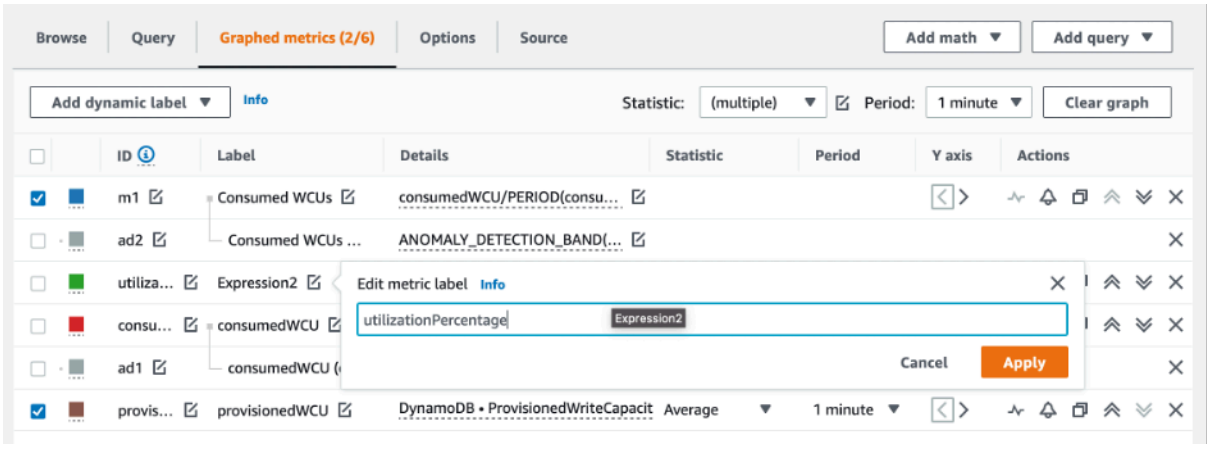
11. Select the **Expression1** label and update the value to **m1** and the label to **Consumed WCUs**.

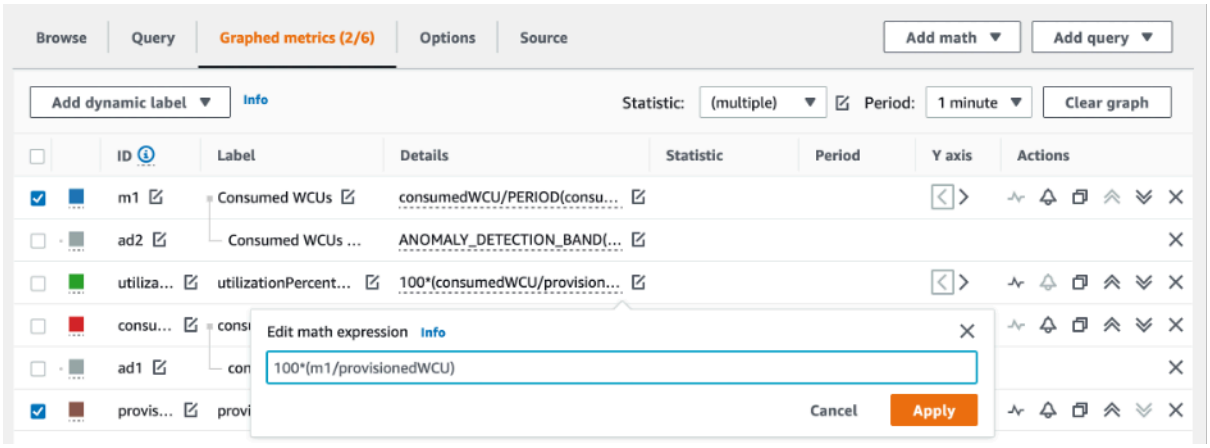
> **Note**
>
> Make sure you have only selected **m1** (checkbox on the left) and **provisionedWCU** to properly visualize the data. Update the formula by clicking in **Details** and changing the formula to **consumedWCU/PERIOD(consumedWCU)**. This step might also generate another **ANOMALY_DETECTION_BAND** metric, but for the scope of this procedure we can ignore it.
>
> 
>

12. You should have now have two graphics: one that indicates your provisioned WCUs on the table and another that indicates the consumed WCUs. The shape of the graphic might be different from the one below, but you can use it as reference:



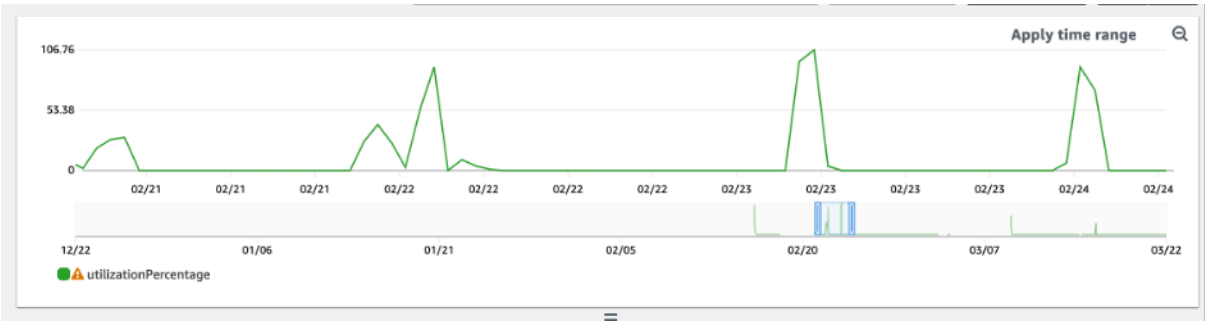
13. Update the percentage formula by selecting the Expression2 graphic (**e2**). Rename the labels and IDs to **utilizationPercentage**. Rename the formula to match **100*(m1/provisionedWCU)**.





14. Remove the checkbox from all the metrics but **utilizationPercentage** to visualize your utilization patterns. The default interval is set to 1 minute, but feel free to modify it as you need.

Here is view of a longer period of time as well as a bigger period of 1 hour. You can see there are some intervals where the utilization was higher than 100%, but this particular workload has longer intervals with zero utilization.



At this point, you might have different results from the pictures in this example. It all depends on the data from your workload. Intervals with more than 100% utilization are prone to throttling events. DynamoDB offers burst capacity, but as soon as the burst capacity is done anything above 100% will be throttled.

How to identify under-provisioned DynamoDB tables

For most workloads, a table is considered under-provisioned when it constantly consumes more than 80% of their provisioned capacity.

Burst capacity is a DynamoDB feature that allow customers to temporarily consume more RCUs/WCUs than originally provisioned (more than the per-second provisioned throughput that was defined in the table). The burst capacity was created to absorb sudden increases in traffic due to special events or usage spikes. This burst capacity doesn't last forever. As soon as the unused RCUs and WCUs are depleted, you will get throttled if you try to consume more capacity than provisioned. When your application traffic is getting close to the 80% utilization rate, your risk of throttling is significantly higher.

The 80% utilization rate rule varies from the seasonality of your data and your traffic growth. Consider the following scenarios:

- If your traffic has been **stable** at ~90% utilization rate for the last 12 months, your table has just the right capacity
- If your application traffic is **growing** at a rate of 8% monthly in less than 3 months, you will arrive at 100%
- If your application traffic is **growing** at a rate of 5% in a little more than 4 months, you will still arrive at 100%

The results from the queries above provide a picture of your utilization rate. Use them as a guide to further evaluate other metrics that can help you choose to increase your table capacity as required (for example: a monthly or weekly growth rate). Work with your operations team to define what is a good percentage for your workload and your tables.

There are special scenarios where the data is skewed when we analyse it on a daily or weekly basis. For example, with seasonal applications that have spikes in usage during working hours (but then drops to almost zero outside of working hours), you could benefit by scheduling auto scaling where you specify the hours of the day (and the days of the week) to increase the provisioned capacity and when to reduce it. Instead of aiming for higher capacity so you can cover the busy hours, you can also benefit from DynamoDB table auto scaling configurations if your seasonality is less pronounced.

> **Note**
>
> When you create a DynamoDB auto scaling configuration for your base table, remember to include another configuration for any GSI that is associated with the table.

How to identify over-provisioned DynamoDB tables

The query results obtained from the scripts above provide the data points required to perform some initial analysis. If your data set presents values lower than 20% utilization for several intervals, your table might be over-provisioned. To further define if you need to reduce the number of WCUs and RCUS, you should revisit the other readings in the intervals.

When your tables contain several low usage intervals, you can really benefit from using auto scaling policies, either by scheduling auto scaling or just configuring the default auto scaling policies for the table that are based on utilization.

If you have a workload with low utilization to high throttle ratio (**Max(ThrottleEvents)/ Min(ThrottleEvents)** in the interval), this could happen when you have a very spiky workload where traffic increases a lot during some days (or hours), but in general the traffic is consistently low. In these scenarios it might be beneficial to use scheduled auto scaling.

The AWS Well-Architected Framework helps cloud architects build secure, high-performing, resilient, and efficient infrastructure for a variety of applications and workloads. Built around six pillars—operational excellence, security, reliability, performance efficiency, cost optimization, and sustainability—AWS Well-Architected provides a consistent approach for customers and partners to evaluate architectures and implement scalable designs.

The AWS Well-Architected Lenses extend the guidance offered by AWS Well-Architected to specific industry and technology domains. The Amazon DynamoDB Well-Architected Lens focuses on DynamoDB workloads. It provides best practices, design principles and questions to assess and review a DynamoDB workload. Completing an Amazon DynamoDB Well-Architected Lens review will provide you with education and guidance around recommended design principles as it relates to each of the AWS Well-Architected pillars. This guidance is based on our experience working with customers across various industries, segments, sizes and geographies.

As a direct outcome of the Well-Architected Lens review, you will receive a summary of actionable recommendations to optimize and improve your DynamoDB workload.

Conducting the Amazon DynamoDB Well-Architected Lens review

The DynamoDB Well-Architected Lens review is usually performed by an AWS Solutions Architect together with the customer, but can also be performed by the customer as a self-service. While we recommend reviewing all six of the Well-Architected Pillars as part of the Amazon DynamoDB Well-Architected Lens, you can also decide to prioritize your focus on one or more pillars first.

Additional information and instructions for conducting an Amazon DynamoDB Well-Architected Lens review are available in this video and the DynamoDB Well-Architected Lens GitHub page.

The pillars of the Amazon DynamoDB Well-Architected Lens

The Amazon DynamoDB Well-Architected Lens is built around six pillars:

Performance efficiency pillar

The performance efficiency pillar includes the ability to use computing resources efficiently to meet system requirements, and to maintain that efficiency as demand changes and technologies evolve.

The primary DynamoDB design principles for this pillar revolve around modeling the data, choosing partition keys and sort keys, and defining secondary indexes based on the application access patterns. Additional considerations include choosing the optimal throughput mode for the workload, AWS SDK tuning and, when appropriate, using an optimal caching strategy. To learn more about these design principles, watch this deep dive video about the performance efficiency pillar of the DynamoDB Well-Architected Lens.

Cost optimization pillar

The cost optimization pillar focuses on avoiding unnecessary costs.

Key topics include understanding and controlling where money is being spent, selecting the most appropriate and right number of resource types, analyzing spend over time, designing your data models to optimize the cost for application-specific access patterns, and scaling to meet business needs without overspending.

The key cost optimization design principles for DynamoDB revolve around choosing the most appropriate capacity mode and table class for your tables and avoiding over-provisioning capacity by either using the on-demand capacity mode, or provisioned capacity mode with autoscaling. Additional considerations include efficient data modeling and querying to reduce the amount of

consumed capacity, reserving portions of the consumed capacity at discounted price, minimizing item size, identifying and removing unused resources and using TTL to automatically delete aged-out data at no cost. To learn more about these design principles, watch this deep dive video about the cost optimization pillar of the DynamoDB Well-Architected Lens.

See Cost optimization for additional information on cost optimization best practices for DynamoDB.

Operational excellence pillar

The operational excellence pillar focuses on running and monitoring systems to deliver business value, and continually improving processes and procedures. Key topics include automating changes, responding to events, and defining standards to manage daily operations.

The main operational excellence design principles for DynamoDB include monitoring DynamoDB metrics through Amazon CloudWatch and AWS Config and automatically alert and remediate when predefined thresholds are breached, or non compliant rules are detected. Additional considerations are defining DynamoDB resources via infrastructure as a code and leveraging tags for better organization, identification and cost accounting of your DynamoDB resources. To learn more about these design principles, watch this deep dive video about the operational excellence pillar of the DynamoDB Well-Architected Lens.

Reliability pillar

The reliability pillar focuses on ensuring a workload performs its intended function correctly and consistently when it's expected to. A resilient workload quickly recovers from failures to meet business and customer demand. Key topics include distributed system design, recovery planning, and how to handle change.

The essential reliability design principles for DynamoDB revolve around choosing the backup strategy and retention based on your RPO and RTO requirements, using DynamoDB global tables for multi-regional workloads, or cross-region disaster recovery scenarios with low RTO, implementing retry logic with exponential backoff in the application by configuring and using these capabilities in the AWS SDK, and monitoring DynamoDB metrics through Amazon CloudWatch and automatically alerting and remediating when predefined thresholds are breached. To learn more about these design principles, watch this deep dive video about the reliability pillar of the DynamoDB Well-Architected Lens.

Security pillar

The security pillar focuses on protecting information and systems. Key topics include confidentiality and integrity of data, identifying and managing who can do what with privilege management, protecting systems, and establishing controls to detect security events.

The main security design principles for DynamoDB are encrypting data in transit with HTTPS, choosing the type of keys for data at rest encryption and defining the IAM roles and policies to authenticate, authorize and provide fine grain access to DynamoDB resources. Additional considerations include auditing DynamoDB control plane and data plane operations through AWS CloudTrail. To learn more about these design principles, watch this deep dive video about the security pillar of the DynamoDB Well-Architected Lens.

See Security for additional information on security for DynamoDB.

Sustainability pillar

The sustainability pillar focuses on minimizing the environmental impacts of running cloud workloads. Key topics include a shared responsibility model for sustainability, understanding impact, and maximizing utilization to minimize required resources and reduce downstream impacts.

The main sustainability design principles for DynamoDB include identifying and removing unused DynamoDB resources, avoiding over-provisioning though the usage of on-demand capacity mode or provisioned capacity-mode with autoscaling, efficient querying to reduce the amount of capacity being consumed and reduction of the storage footprint by compressing data and by deleting aged-out data through the use of TTL. To learn more about these design principles, watch this deep dive video about the sustainability pillar of the DynamoDB Well-Architected Lens.

Best practices for designing and using partition keys effectively in DynamoDB

The primary key that uniquely identifies each item in an Amazon DynamoDB table can be simple (a partition key only) or composite (a partition key combined with a sort key).

You should design your application for uniform activity across all partition keys in the table and its secondary indexes. You can determine the access patterns that your application requires, and read and write units that each table and secondary index requires.

> **Note**
>
> Adaptive capacity applies to on-demand mode and provisioned capacity.

Every partition in a DynamoDB table is designed to deliver a maximum capacity of 3,000 read units per second and 1,000 write units per second. One read unit represents one strongly consistent read operation per second, or two eventually consistent read operations per second, for an item up to 4 KB in size. One write unit represents one write operation per second for an item up to 1 KB in size.

You must factor in the item size when evaluating the partition throughput limits for your table. For example, if the table has an item size of 20 KB, a single consistent read operation will consume 5 read units. This means you can concurrently drive 600 consistent read operations per second on that single item before reaching the partition limits. The total throughput across all partitions in the table can be constrained by the provisioned throughput in provisioned mode, or by the table level throughput limit in on-demand mode. See Service Quotas for more information.

Topics

- Designing partition keys to distribute your workload in DynamoDB
- Using write sharding to distribute workloads evenly in your DynamoDB table
- Distributing write activity efficiently during data upload in DynamoDB

Designing partition keys to distribute your workload in DynamoDB

The partition key portion of a table's primary key determines the logical partitions in which a table's data is stored. This in turn affects the underlying physical partitions. A partition key design that doesn't distribute I/O requests effectively can create "hot" partitions that result in throttling and use your provisioned I/O capacity inefficiently.

The optimal usage of a table's provisioned throughput depends not only on the workload patterns of individual items, but also on the partition key design. This doesn't mean that you must access all partition key values to achieve an efficient throughput level, or even that the percentage of accessed partition key values must be high. It does mean that the more distinct partition key values that your workload accesses, the more those requests will be spread across the partitioned space. In general, you'll use your provisioned throughput more efficiently as the ratio of partition key values accessed to the total number of partition key values increases.

The following is a comparison of the provisioned throughput efficiency of some common partition key schemas.

Partition key value	Uniformity
User ID, where the application has many users.	Good
Status code, where there are only a few possible status codes.	Bad
Item creation date, rounded to the nearest time period (for example, day, hour, or minute).	Bad
Device ID, where each device accesses data at relatively similar intervals.	Good
Device ID, where even if there are many devices being tracked, one is by far more popular than all the others.	Bad

If a single table has only a small number of partition key values, consider distributing your write operations across more distinct partition key values. In other words, structure the primary key elements to avoid one "hot" (heavily requested) partition key value that slows overall performance.

For example, consider a table with a composite primary key. The partition key represents the item's creation date, rounded to the nearest day. The sort key is an item identifier. On a given day, say `2014-07-09`, **all** of the new items are written to that single partition key value (and corresponding physical partition).

If the table fits entirely into a single partition (considering growth of your data over time), and if your application's read and write throughput requirements don't exceed the read and write capabilities of a single partition, your application won't encounter any unexpected throttling as a result of partitioning.

To use NoSQL Workbench for DynamoDB to help visualize your partition key design, see Building data models with NoSQL Workbench.

Using write sharding to distribute workloads evenly in your DynamoDB table

One way to better distribute writes across a partition key space in Amazon DynamoDB is to expand the space. You can do this in several different ways. You can add a random number to the partition key values to distribute the items among partitions. Or you can use a number that is calculated based on something that you're querying on.

Sharding using random suffixes

One strategy for distributing loads more evenly across a partition key space is to add a random number to the end of the partition key values. Then you randomize the writes across the larger space.

For example, for a partition key that represents today's date, you might choose a random number between 1 and 200 and concatenate it as a suffix to the date. This yields partition key values like 2014-07-09.1, 2014-07-09.2, and so on, through 2014-07-09.200. Because you are randomizing the partition key, the writes to the table on each day are spread evenly across multiple partitions. This results in better parallelism and higher overall throughput.

However, to read all the items for a given day, you would have to query the items for all the suffixes and then merge the results. For example, you would first issue a Query request for the partition key value 2014-07-09.1. Then issue another Query for 2014-07-09.2, and so on, through 2014-07-09.200. Finally, your application would have to merge the results from all those Query requests.

Sharding using calculated suffixes

A randomizing strategy can greatly improve write throughput. But it's difficult to read a specific item because you don't know which suffix value was used when writing the item. To make it easier to read individual items, you can use a different strategy. Instead of using a random number to distribute the items among partitions, use a number that you can calculate based upon something that you want to query on.

Consider the previous example, in which a table uses today's date in the partition key. Now suppose that each item has an accessible OrderId attribute, and that you most often need to find items by order ID in addition to date. Before your application writes the item to the table, it could calculate a hash suffix based on the order ID and append it to the partition key date. The calculation might

generate a number between 1 and 200 that is fairly evenly distributed, similar to what the random strategy produces.

A simple calculation would likely suffice, such as the product of the UTF-8 code point values for the characters in the order ID, modulo 200, + 1. The partition key value would then be the date concatenated with the calculation result.

With this strategy, the writes are spread evenly across the partition key values, and thus across the physical partitions. You can easily perform a `GetItem` operation for a particular item and date because you can calculate the partition key value for a specific `OrderId` value.

To read all the items for a given day, you still must `Query` each of the `2014-07-09.N` keys (where `N` is 1–200), and your application then has to merge all the results. The benefit is that you avoid having a single "hot" partition key value taking all of the workload.

> **Note**
>
> For a more efficient strategy specifically designed to handle high-volume time series data, see Time series data.

Distributing write activity efficiently during data upload in DynamoDB

Typically, when you load data from other data sources, Amazon DynamoDB partitions your table data on multiple servers. You get better performance if you upload data to all the allocated servers simultaneously.

For example, suppose that you want to upload user messages to a DynamoDB table that uses a composite primary key with `UserID` as the partition key and `MessageID` as the sort key.

When you upload the data, one approach you can take is to upload all message items for each user, one user after another:

UserID	MessageID
U1	1
U1	2

UserID	MessageID
U1	...
U1	... up to 100
U2	1
U2	2
U2	...
U2	... up to 200

The problem in this case is that you are not distributing your write requests to DynamoDB across your partition key values. You are taking one partition key value at a time and uploading all of its items before going to the next partition key value and doing the same.

Behind the scenes, DynamoDB is partitioning the data in your table across multiple servers. To fully use all the throughput capacity that is provisioned for the table, you must distribute your workload across your partition key values. By directing an uneven amount of upload work toward items that all have the same partition key value, you are not fully using all the resources that DynamoDB has provisioned for your table.

You can distribute your upload work by using the sort key to load one item from each partition key value, then another item from each partition key value, and so on:

UserID	MessageID
U1	1
U2	1
U3	1
...	...
U1	2
U2	2

UserID	MessageID
U3	2
...	...

Every upload in this sequence uses a different partition key value, keeping more DynamoDB servers busy simultaneously and improving your throughput performance.

Best practices for using sort keys to organize data in DynamoDB

In an Amazon DynamoDB table, the primary key that uniquely identifies each item in the table can be composed of a partition key and a sort key.

Well-designed sort keys have two key benefits:

- They gather related information together in one place where it can be queried efficiently. Careful design of the sort key lets you retrieve commonly needed groups of related items using range queries with operators such as `begins_with`, `between`, >, <, and so on.
- Composite sort keys let you define hierarchical (one-to-many) relationships in your data that you can query at any level of the hierarchy.

  For example, in a table listing geographical locations, you might structure the sort key as follows.

  ```
  [country]#[region]#[state]#[county]#[city]#[neighborhood]
  ```

  This would let you make efficient range queries for a list of locations at any one of these levels of aggregation, from `country`, to a `neighborhood`, and everything in between.

Using sort keys for version control

Many applications need to maintain a history of item-level revisions for audit or compliance purposes and to be able to retrieve the most recent version easily. There is an effective design pattern that accomplishes this using sort key prefixes:

- For each new item, create two copies of the item: One copy should have a version-number prefix of zero (such as `v0_`) at the beginning of the sort key, and one should have a version-number prefix of one (such as `v1_`).
- Every time the item is updated, use the next higher version-prefix in the sort key of the updated version, and copy the updated contents into the item with version-prefix zero. This means that the latest version of any item can be located easily using the zero prefix.

For example, a parts manufacturer might use a schema like the one illustrated below.

Primary Key		Data-Item Attributes...				
Partition Key *Equipment_ID*	Sort Key *(varies)*	Attribute 1	Attribute 2	Attribute 3	Attribute 4	...
Equipment_1	Details	Name: Biphasic Cardiometer *(equipment name)*	Factory_ID: S14_Tukwilla *(factory where manufactured)*	Line_ID R_7 *(assembly-line ID)*		
	v0_Audit	Auditor: Padma *(name of the auditor)*	Latest: 3 *(most recent audit version)*	Time: 2018-04-15T11:00 *(audit date and time)*	Result Passed *(audit result)*	...etc.
	v1_Audit	Auditor: Rick *(name of the auditor)*	Time: 2018-03-14T11:00 *(audit date and time)*	Result Open *(audit result)*	Report: 0943922EKG14 *(detailed problem report in S3)*	...etc.
	v2_Audit	Auditor: George *(name of the auditor)*	Time: 2018-03-18T11:00 *(audit date and time)*	Result Open *(audit result)*	Report: 0943923EKG15 *(detailed problem report in S3)*	...etc.
	v3_Audit	Auditor: Padma *(name of the auditor)*	Time: 2018-04-15T11:00 *(audit date and time)*	Result Passed *(audit result)*	Report: x792 *(pass confirmation report)*	...etc.

The `Equipment_1` item goes through a sequence of audits by various auditors. The results of each new audit are captured in a new item in the table, starting with version number one, and then incrementing the number for each successive revision.

When each new revision is added, the application layer replaces the contents of the zero-version item (having sort key equal to `v0_Audit`) with the contents of the new revision.

Whenever the application needs to retrieve for the most recent audit status, it can query for the sort key prefix of `v0_`.

If the application needs to retrieve the entire revision history, it can query all the items under the item's partition key and filter out the `v0_` item.

This design also works for audits across multiple parts of a piece of equipment, if you include the individual part-IDs in the sort key after the sort key prefix.

Best practices for using secondary indexes in DynamoDB

Secondary indexes are often essential to support the query patterns that your application requires. At the same time, overusing secondary indexes or using them inefficiently can add cost and reduce performance unnecessarily.

Contents

- General guidelines for secondary indexes in DynamoDB
  - Use indexes efficiently
  - Choose projections carefully
  - Optimize frequent queries to avoid fetches
  - Be aware of item-collection size limits when creating local secondary indexes
- Take advantage of sparse indexes
  - Examples of sparse indexes in DynamoDB
- Using Global Secondary Indexes for materialized aggregation queries in DynamoDB
  - Example scenario and access patterns
  - Why pre-compute aggregations
  - Table design
  - Aggregation pipeline with Streams and AWS Lambda
  - Sparse GSI design
  - Querying the GSI
  - Considerations
- Overloading Global Secondary Indexes in DynamoDB
- Using Global Secondary Index write sharding for selective table queries in DynamoDB
  - Pattern design
  - Sharding strategy
  - Querying the sharded GSI
  - Parallel query execution considerations
  - Code example
- Using Global Secondary Indexes to create an eventually consistent replica in DynamoDB

General guidelines for secondary indexes in DynamoDB

Amazon DynamoDB supports two types of secondary indexes:

- **Global secondary index (GSI)—** An index with a partition key and a sort key that can be different from those on the base table. A global secondary index is considered "global" because queries on the index can span all of the data in the base table, across all partitions. A global secondary index has no size limitations and has its own provisioned throughput settings for read and write activity that are separate from those of the table.
- **Local secondary index (LSI)—**An index that has the same partition key as the base table, but a different sort key. A local secondary index is "local" in the sense that every partition of a local secondary index is scoped to a base table partition that has the same partition key value. As a result, the total size of indexed items for any one partition key value can't exceed 10 GB. Also, a local secondary index shares provisioned throughput settings for read and write activity with the table it is indexing.

Each table in DynamoDB can have up to 20 global secondary indexes (default quota) and 5 local secondary indexes.

Global secondary indexes are often more useful than local secondary indexes. Determining which type of index to use will also depend on your application's requirements. For a comparison of global secondary indexes and local secondary indexes, and more information on how to choose between them, see the section called “Working with indexes”.

The following are some general principles and design patterns to keep in mind when creating indexes in DynamoDB:

Topics

- Use indexes efficiently
- Choose projections carefully
- Optimize frequent queries to avoid fetches
- Be aware of item-collection size limits when creating local secondary indexes

Use indexes efficiently

Keep the number of indexes to a minimum. Don't create secondary indexes on attributes that you don't query often. Indexes that are seldom used contribute to increased storage and I/O costs without improving application performance.

Choose projections carefully

Because secondary indexes consume storage and provisioned throughput, you should keep the size of the index as small as possible. Also, the smaller the index, the greater the performance advantage compared to querying the full table. If your queries usually return only a small subset of attributes, and the total size of those attributes is much smaller than the whole item, project only the attributes that you regularly request.

If you expect a lot of write activity on a table compared to reads, follow these best practices:

- Consider projecting fewer attributes to minimize the size of items written to the index. However, this only applies if the size of projected attributes would otherwise be larger than a single write capacity unit (1 KB). For example, if the size of an index entry is only 200 bytes, DynamoDB rounds this up to 1 KB. In other words, as long as the index items are small, you can project more attributes at no extra cost.
- Avoid projecting attributes that you know will rarely be needed in queries. Every time you update an attribute that is projected in an index, you incur the extra cost of updating the index as well. You can still retrieve non-projected attributes in a `Query` at a higher provisioned throughput cost, but the query cost may be significantly lower than the cost of updating the index frequently.
- Specify `ALL` only if you want your queries to return the entire table item sorted by a different sort key. Projecting all attributes eliminates the need for table fetches, but in most cases, it doubles your costs for storage and write activity.

Balance the need to keep your indexes as small as possible against the need to keep fetches to a minimum, as explained in the next section.

Optimize frequent queries to avoid fetches

To get the fastest queries with the lowest possible latency, project all the attributes that you expect those queries to return. In particular, if you query a local secondary index for attributes that are not

projected, DynamoDB automatically fetches those attributes from the table, which requires reading the entire item from the table. This introduces latency and additional I/O operations that you can avoid.

Keep in mind that "occasional" queries can often turn into "essential" queries. If there are attributes that you don't intend to project because you anticipate querying them only occasionally, consider whether circumstances might change and you might regret not projecting those attributes after all.

For more information about table fetches, see Provisioned throughput considerations for Local Secondary Indexes.

Be aware of item-collection size limits when creating local secondary indexes

An *item collection* is all the items in a table and its local secondary indexes that have the same partition key. No item collection can exceed 10 GB, so it's possible to run out of space for a particular partition key value.

When you add or update a table item, DynamoDB updates all local secondary indexes that are affected. If the indexed attributes are defined in the table, the local secondary indexes grow too.

When you create a local secondary index, think about how much data will be written to it, and how many of those data items will have the same partition key value. If you expect that the sum of table and index items for a particular partition key value might exceed 10 GB, consider whether you should avoid creating the index.

If you can't avoid creating the local secondary index, you must anticipate the item collection size limit and take action before you exceed it. As a best practice, you should utilize the `ReturnItemCollectionMetrics` parameter when writing items to monitor and alert on item collection sizes that approach the 10GB size limit. Exceeding the maximum item collection size will result in failed write attempts. You can mitigate the item collection size issues by monitoring and alerting on item collection sizes before they impact your application.

> **Note**
>
> Once created, you cannot delete a local secondary index.

For strategies on working within the limit and taking corrective action, see Item collection size limit.

Take advantage of sparse indexes

For any item in a table, DynamoDB writes a corresponding index entry **only if the index key attributes are present in the item**. For a global secondary index, this means the index partition key must be defined on the item, and if the index also has a sort key, that attribute must be present too. If either key attribute is missing from an item, that item does not appear in the index. An index where only a subset of items from the base table appear is called a *sparse* index.

Sparse indexes are useful for queries over a small subsection of a table. For example, suppose that you have a table where you store all your customer orders, with the following key attributes:

- Partition key: `CustomerId`
- Sort key: `OrderId`

To track open orders, you can insert an attribute named `isOpen` in order items that have not already shipped. Then when the order ships, you can delete the attribute. If you then create an index on `CustomerId` (partition key) and `isOpen` (sort key), only those orders with `isOpen` defined appear in it. When you have thousands of orders of which only a small number are open, it's faster and less expensive to query that index for open orders than to scan the entire table.

Instead of using a type of attribute like `isOpen`, you could use an attribute with a value that results in a useful sort order in the index. For example, you could use an `OrderOpenDate` attribute set to the date on which each order was placed, and then delete it after the order is fulfilled. That way, when you query the sparse index, the items are returned sorted by the date on which each order was placed.

Examples of sparse indexes in DynamoDB

Global secondary indexes are sparse by default. When you create a global secondary index, you specify a partition key and optionally a sort key. Only items in the base table that contain the required key attributes appear in the index. If an item is missing the index partition key—or the sort key, when one is defined—that item is excluded from the index.

By designing a global secondary index to be sparse, you can provision it with lower write throughput than that of the base table, while still achieving excellent performance.

For example, a gaming application might track all scores of every user, but generally only needs to query a few high scores. The following design handles this scenario efficiently:

Table

Primary Key		Data Attributes...		
Partition Key	*Sort Key*			
Player_ID	Game_ID	Attribute 1	Attribute 2	Attribute 3
Rick	Game_1	Score: 36,750 *(game score)*	Date: 2017-11-14 *(date of game)*	
	Game_2	Score: 69,450 *(game score)*	Date: 2017-12-31 *(date of game)*	
	Game_3	Score: 135,900 *(game score)*	Date: 2018-01-19 *(date of game)*	Award: Champ *(type of award)*
Padma	Game_4	Score: 25,350 *(game score)*	Date: 2018-01-27 *(date of game)*	
	Game_5	Score: 69,450 *(game score)*	Date: 2028-01-19 *(date of game)*	
	Game_6	Score: 147,300 *(game score)*	Date: 2018-02-02 *(date of game)*	Award: Champ *(type of award)*
	Game_7	Score: 169,100 *(game score)*	Date: 2018-03-10 *(date of game)*	Award: Champ *(type of award)*

Here, Rick has played three games and achieved `Champ` status in one of them. Padma has played four games and achieved `Champ` status in two of them. Notice that the `Award` attribute is present only in items where the user achieved an award. The associated global secondary index looks like the following:

GSI

Primary Key	Projected Attributes...			
Partition Key				
Award	Player_ID	Game_ID	Score	Date
Champ	Rick	Game_3	135,900	2018-01-19
	Padma	Game_6	147,300	2018-02-02
	Padma	Game_7	169,100	2018-03-10

The global secondary index contains only the high scores that are frequently queried, which are a small subset of the items in the base table.

Using Global Secondary Indexes for materialized aggregation queries in DynamoDB

Maintaining near real-time aggregations and key metrics on top of rapidly changing data is becoming increasingly valuable to businesses for making rapid decisions. For example, a music

library might want to showcase its most downloaded songs in near-real time, or an e-commerce platform might need to display trending products by category.

Because DynamoDB doesn't natively support aggregation operations like `SUM` or `COUNT` across items, computing these values at read time would require scanning large numbers of items—which may be slow and expensive. Instead, you can *pre-compute* aggregations as data changes and store the results as regular items in your table. This pattern is called *materialized aggregation*.

Topics

- Example scenario and access patterns
- Why pre-compute aggregations
- Table design
- Aggregation pipeline with Streams and AWS Lambda
- Sparse GSI design
- Querying the GSI
- Considerations

Example scenario and access patterns

Consider a music library application with the following requirements:

- The application records individual song downloads at high volume (thousands per second).
- Users need to see the most downloaded songs for a given month with single-digit millisecond latency.
- The application also needs to support queries like "top 10 songs this month" and "all songs downloaded in a given month."

Computing download counts at read time by scanning all download records may be expensive at this scale. Instead, you can maintain a running count that updates as each download occurs, and store it in a way that supports efficient querying.

Why pre-compute aggregations

There are several approaches to computing aggregations. The following table compares common alternatives and explains why materialized aggregation in DynamoDB is often the best fit for this type of use case.

Approach	Tradeoffs	When to use
Scan and count at read time	Requires reading all download records for every query. Latency grows with data volume and consumes significant read capacity.	Only suitable for very small datasets where latency isn't a concern.
External aggregati on store (for example, Amazon ElastiCache)	Adds operational complexity with a separate service to manage. Requires synchronization logic between DynamoDB and the cache.	When you need sub-millisecond reads or complex aggregation logic that goes beyond simple counts.
Application-level aggregation on write	Couples the aggregation logic to the write path. If the application fails after recording the download but before updating the count, the aggregation becomes inconsistent.	When you need synchronous, strongly consistent aggregation and can tolerate added write latency.
Materialized aggregation with Streams and Lambda	Decouples aggregation from the write path. Aggregation is eventuall y consistent (typically seconds behind). Adds Lambda invocation costs.	**When you need near real-time aggregations with low read latency and can tolerate eventual consisten cy.** This is the approach described on this page.

The materialized aggregation approach keeps the write path simple (just record the download), offloads the aggregation to an asynchronous process, and stores the result in DynamoDB where it can be queried with single-digit millisecond latency.

Table design

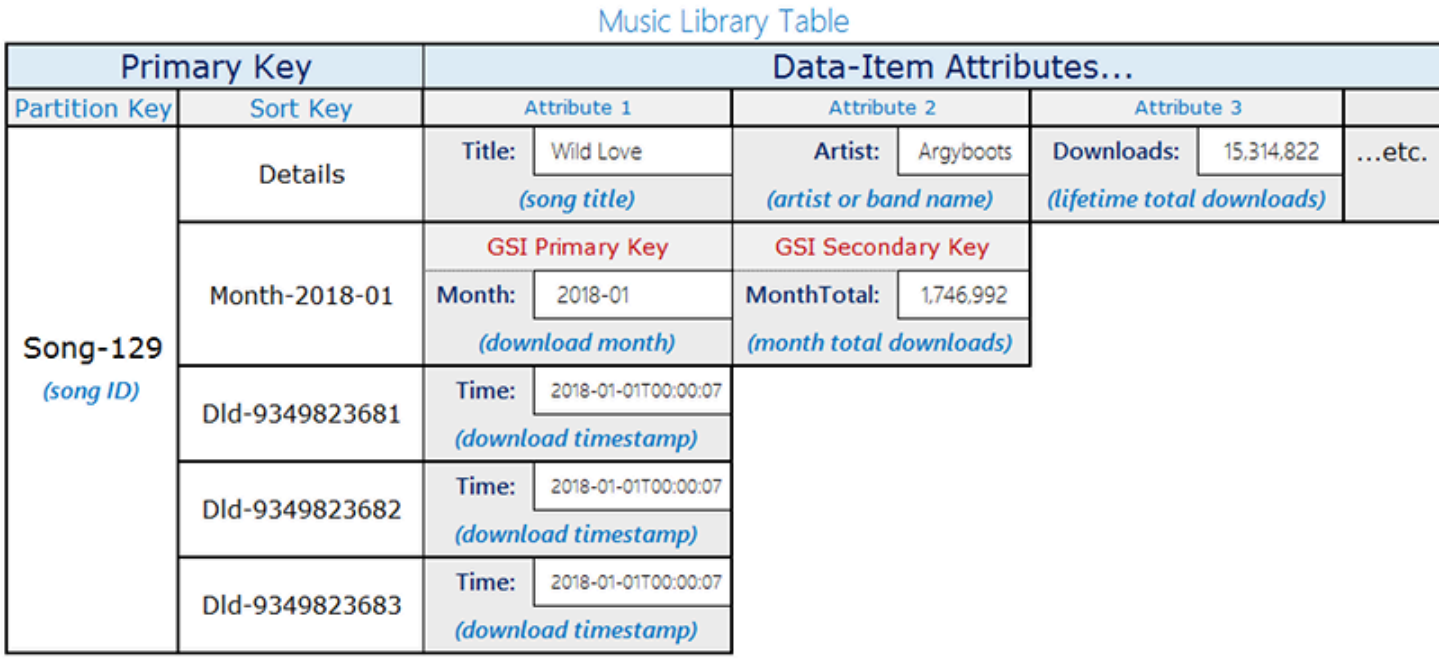
This design uses a single table with two item types that share the same partition key (`songID`) but use different sort key patterns to distinguish between them:

- **Download records** – Individual download events. The sort key is the `DownloadID` (a unique identifier for each download).

- **Monthly aggregation items** – Pre-computed download counts per song per month. The sort key is the month in `YYYY-MM` format (for example, `2018-01`). These items also contain a `DownloadCount` attribute with the running total.

Only the monthly aggregation items contain the `Month` attribute. This distinction is important for the sparse GSI design described later.

The following diagram shows the table layout with both item types:



Item type	Partition key (songID)	Sort key	Additional attributes
Download record	`song1`	`download-abc123`	`UserID`, `Timestamp`
Monthly aggregation	`song1`	`2018-01`	`Month=2018-01`, `DownloadCount =1,746,992`

Aggregation pipeline with Streams and AWS Lambda

The aggregation pipeline works as follows:

1. When a song is downloaded, the application writes a new item to the table with `Partition-Key=songID` and `Sort-Key=DownloadID`.
2. DynamoDB Streams captures this write as a stream record.
3. A Lambda function, attached to the stream, processes the new record. It identifies the `songID` and the current month, then updates the corresponding monthly aggregation item by incrementing the `DownloadCount` attribute.
4. The updated aggregation item is then available for querying through the sparse GSI.

The Lambda function uses an `UpdateItem` call with an `ADD` expression to atomically increment the download count. This avoids read-modify-write race conditions:

```
import boto3

dynamodb = boto3.resource('dynamodb')
table = dynamodb.Table('MusicLibrary')

def handler(event, context):
    for record in event['Records']:
        if record['eventName'] == 'INSERT':
            new_image = record['dynamodb']['NewImage']
            song_id = new_image['songID']['S']
            # Derive the month from the download timestamp
            timestamp = new_image['Timestamp']['S']
            month = timestamp[:7]  # Extract YYYY-MM

            table.update_item(
                Key={
                    'songID': song_id,
                    'SK': month
                },
                UpdateExpression='ADD DownloadCount :inc SET #m = :month',
                ExpressionAttributeNames={
                    '#m': 'Month'
                },
                ExpressionAttributeValues={
                    ':inc': 1,
                    ':month': month
                }
            )
```

Note

If a Lambda execution fails after writing the updated aggregation value, the stream record may be retried. Because the ADD operation increments the count each time it runs, a retry would increment the count more than once for the same download, leaving you with an *approximate* value. For most analytics and leaderboard use cases, this small margin of error is acceptable. If you need exact counts, consider adding idempotency logic—for example, by using a condition expression that checks whether the specific `DownloadID` has already been processed.

Sparse GSI design

To efficiently query the aggregated results, create a global secondary index with the following key schema:

- **GSI partition key:** `Month` (String)
- **GSI sort key:** `DownloadCount` (Number)

This GSI is *sparse* because only the monthly aggregation items contain the `Month` attribute. The individual download records don't have this attribute, so they are automatically excluded from the index. This means the GSI contains only the pre-computed aggregation items—a small fraction of the total items in the table.

A sparse GSI provides two key benefits:

- **Lower cost** – Because only aggregation items are replicated to the index, you consume far less write capacity and storage compared to an index that includes every item in the table.
- **Faster queries** – The index contains only the data you need to query, so reads are efficient and return results with single-digit millisecond latency.

For more information about how sparse indexes work, see Take advantage of sparse indexes.

Querying the GSI

With the sparse GSI in place, you can efficiently answer several types of queries:

Get the most downloaded song for a given month:

```
aws dynamodb query \
    --table-name "MusicLibrary" \
    --index-name "MonthDownloadsIndex" \
    --key-condition-expression "#m = :month" \
    --expression-attribute-names '{"#m": "Month"}' \
    --expression-attribute-values '{":month": {"S": "2018-01"}}' \
    --scan-index-forward false \
    --limit 1
```

Setting `ScanIndexForward` to `false` sorts results by `DownloadCount` in descending order, and `Limit=1` returns only the top song.

Get the top 10 songs for a given month:

```
aws dynamodb query \
    --table-name "MusicLibrary" \
    --index-name "MonthDownloadsIndex" \
    --key-condition-expression "#m = :month" \
    --expression-attribute-names '{"#m": "Month"}' \
    --expression-attribute-values '{":month": {"S": "2018-01"}}' \
    --scan-index-forward false \
    --limit 10
```

Get all songs downloaded in a given month (sorted by download count):

```
aws dynamodb query \
    --table-name "MusicLibrary" \
    --index-name "MonthDownloadsIndex" \
    --key-condition-expression "#m = :month" \
    --expression-attribute-names '{"#m": "Month"}' \
    --expression-attribute-values '{":month": {"S": "2018-01"}}' \
    --scan-index-forward false
```

Considerations

Keep the following in mind when implementing this pattern:

- **Eventual consistency** – The aggregation values are updated asynchronously through DynamoDB Streams and Lambda. There is typically a delay of a few seconds between a download being recorded and the aggregation being updated. This means the GSI reflects near real-time data, not real-time data.

- **Lambda concurrency** – If your table has a high write volume, multiple Lambda invocations may attempt to update the same aggregation item concurrently. The atomic ADD operation handles this safely, but you should monitor Lambda concurrency and throttling metrics to ensure your function can keep up with the stream.
- **GSI write capacity** – Because the sparse GSI only contains aggregation items, it requires significantly less write capacity than the base table. However, you should still provision enough capacity (or use on-demand mode) to handle the rate of aggregation updates.
- **Approximate counts** – As noted earlier, Lambda retries can cause counts to be slightly over-counted. For use cases that require exact counts, implement idempotency checks in the Lambda function.

Overloading Global Secondary Indexes in DynamoDB

Although Amazon DynamoDB has a default quota of 20 global secondary indexes per table, in practice, you can index across far more than 20 data fields. As opposed to a table in a relational database management system (RDBMS), in which the schema is uniform, a table in DynamoDB can hold many different kinds of data items at one time. In addition, the same attribute in different items can contain entirely different kinds of information.

Consider the following example of a DynamoDB table layout that saves a variety of different kinds of data.

Primary Key		Data-Item Attributes...		
Partition Key	Sort Key	Attribute 1	Attribute 2	...
HR-974 *(employee ID)*	Employee_Name	Data: Murphy, John *(employee name)*	Start: 2008-11-08 *(start date)*	...etc.
	YYYY-Q1	Data: $5,477 *(order totals in USD)*	Name: Murphy, John *(employee name)*	
	HR_confidential	Data: 2008-11-08 *(hire date)*	Name: Murphy, John *(employee name)*	...etc.
	Warehouse_01	Data: Murphy, John *(employee name)*		
	v0_Job_title	Data: Operator-1 *(job title)*	Start: 2008-11-08 *(start date)*	...etc.
	v1_Job_title	Data: Operator-2 *(job title)*	Start: 2016-11-04 *(start date)*	...etc.
	v2_Job_title	Data: Supervisor-1 *(job title)*	Start: 2017-11-01 *(start date)*	...etc.

The `Data` attribute, which is common to all the items, has different content depending on its parent item. If you create a global secondary index for the table that uses the table's sort key as its partition key and the `Data` attribute as its sort key, you can make a variety of different queries using that single global secondary index. These queries might include the following:

- Look up an employee by name in the global secondary index, using `Employee_Name` as the partition key value and the employee's name (for example `Murphy, John`) as the sort key value.
- Use the global secondary index to find all employees working in a particular warehouse by searching on a warehouse ID (such as `Warehouse_01`).
- Get a list of recent hires, querying the global secondary index on `HR_confidential` as a partition key value and using a range of dates in the sort key value.

Using Global Secondary Index write sharding for selective table queries in DynamoDB

When you need to query recent data within a specific time window, DynamoDB's requirement of providing a partition key for most read operations can present a challenge. To address this scenario, you can implement an effective query pattern using a combination of write sharding and a Global Secondary Index (GSI).

This approach allows you to efficiently retrieve and analyze time-sensitive data without performing full table scans, which can be resource-intensive and costly. By strategically designing your table structure and indexing, you can create a flexible solution that supports time-based data retrieval while maintaining optimal performance.

Topics

- Pattern design
- Sharding strategy
- Querying the sharded GSI
- Parallel query execution considerations
- Code example

Pattern design

When working with DynamoDB, you can overcome time-based data retrieval challenges by implementing a sophisticated pattern that combines write sharding and Global Secondary Indexes to enable flexible, efficient querying across recent data windows.

Structure of the table

- Partition Key (PK): "Username"

Structure of the GSI

- GSI Partition Key (PK_GSI): "ShardNumber#"
- GSI Sort Key (SK_GSI): ISO 8601 timestamp (e.g., "2030-04-01T12:00:00Z")

[TABLE] - TimeBounded Metadata Actions

PK	SK_GSI	Address	PK_GSI
Trenton69	2023-12-05T06:40:31.312Z	117 Hudson Divide	ShardNumber#0
Scot_Langworth-Prosacco	2024-10-09T14:44:45.819Z	84021 Herzog Skyway	ShardNumber#6
Juliet.Morissette	2025-03-28T07:20:56.538Z	4871 N Broadway	ShardNumber#4
Kay.Von71	2024-11-23T16:19:26.763Z	14554 Odell Throughway	ShardNumber#2
Kiarra43	2023-11-15T16:21:57.078Z	8471 London Road	ShardNumber#1
Marcella91	2024-12-17T09:02:31.623Z	10271 Nick Crescent	ShardNumber#6
Bernard.Lemke36	2025-09-09T00:18:34.482Z	5438 Douglas Groves	ShardNumber#2
Daisha83	2024-01-24T06:38:08.482Z	8786 Armstrong Radial	ShardNumber#3
Marcos_Schmitt58	2024-04-06T17:38:58.551Z	510 S Center Street	ShardNumber#3
Jeremie_VonRueden-Mills	2025-04-06T04:24:23.701Z	157 Koch Drives	ShardNumber#1
Verna28	2025-10-10T18:42:21.059Z	70040 Baylee Trafficway	ShardNumber#1
Trycia.Doyle	2024-01-10T17:00:08.360Z	9511 Tillman Extensions	ShardNumber#2

Sharding strategy

Assuming you decide to use 10 shards, your shard numbers could range from 0 to 9. When logging an activity, you would calculate the shard number (for example, by using a hash function on the user ID and then taking the modulus of the number of shards) and prepend it to the GSI partition key. This method distributes the entries across different shards, mitigating the risk of hot partitions.

Querying the sharded GSI

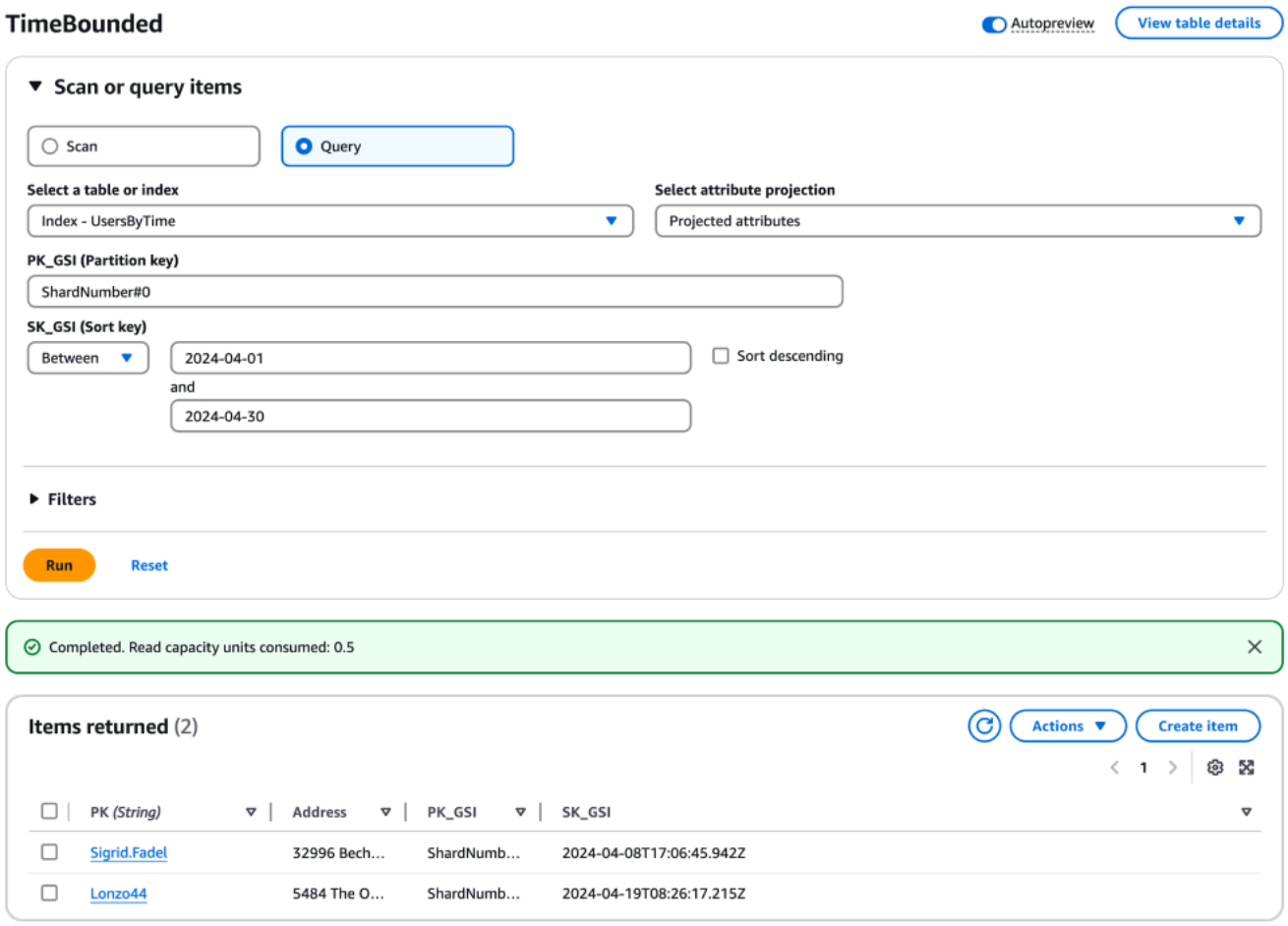
Querying across all shards for items within a particular time range in a DynamoDB table, where data is sharded across multiple partition keys, requires a different approach than querying a single partition. Since DynamoDB queries are limited to a single partition key at a time, you can't directly query across multiple shards with a single query operation. However, you can achieve the desired result through application-level logic by performing multiple queries, each targeting a specific shard, and then aggregating the results. The procedure below explains how to do this.

To query and aggregate shards

1. Identify the range of shard numbers used in your sharding strategy. For instance, if you have 10 shards, your shard numbers would range from 0-9.

2. For each shard, construct and execute a query to fetch items within the desired time range. These queries can be executed in parallel to improve efficiency. Use the partition key with the shard number and the sort key with your time range for these queries. Here's an example query for a single shard:

```
aws dynamodb query \
    --table-name "YourTableName" \
    --index-name "YourIndexName" \
    --key-condition-expression "PK_GSI = :pk_val AND SK_GSI BETWEEN :start_date
 AND :end_date" \
    --expression-attribute-values '{
        ":pk_val": {"S": "ShardNumber#0"},
        ":start_date": {"S": "2024-04-01"},
        ":end_date": {"S": "2024-04-30"}
    }'
```



You would replicate this query for each shard, adjusting the partition key accordingly (e.g., "ShardNumber#1", "ShardNumber#2", ..., "ShardNumber#9").

3. Aggregate the results from each query after all queries are complete. Perform this aggregation in your application code, combining the results into a single dataset that represents the items from all shards within your specified time range.

Parallel query execution considerations

Each query consumes read capacity from your table or index. If you're using provisioned throughput, ensure that your table is provisioned with enough capacity to handle the burst of parallel queries. If you're using on-demand capacity, be mindful of the potential cost implications.

Code example

To execute parallel queries across shards in DynamoDB using Python, you can use the boto3 library, which is the Amazon Web Services SDK for Python. This example assumes you have boto3 installed and configured with appropriate AWS credentials.

The following Python code demonstrates how to perform parallel queries across multiple shards for a given time range. It uses concurrent futures to execute queries in parallel, reducing the overall execution time compared to sequential execution.

```
import boto3
from concurrent.futures import ThreadPoolExecutor, as_completed

# Initialize a DynamoDB client
dynamodb = boto3.client('dynamodb')

# Define your table name and the total number of shards
table_name = 'YourTableName'
total_shards = 10  # Example: 10 shards numbered 0 to 9
time_start = "2030-03-15T09:00:00Z"
time_end = "2030-03-15T10:00:00Z"

def query_shard(shard_number):
    """
    Query items in a specific shard for the given time range.
    """
    response = dynamodb.query(
```

```
        TableName=table_name,
        IndexName='YourGSIName',  # Replace with your GSI name
        KeyConditionExpression="PK_GSI = :pk_val AND SK_GSI BETWEEN :date_start
 AND :date_end",
        ExpressionAttributeValues={
            ":pk_val": {"S": f"ShardNumber#{shard_number}"},
            ":date_start": {"S": time_start},
            ":date_end": {"S": time_end},
        }
    )
    return response['Items']

# Use ThreadPoolExecutor to query across shards in parallel
with ThreadPoolExecutor(max_workers=total_shards) as executor:
    # Submit a future for each shard query
    futures = {executor.submit(query_shard, shard_number): shard_number for
 shard_number in range(total_shards)}

    # Collect and aggregate results from all shards
    all_items = []
    for future in as_completed(futures):
        shard_number = futures[future]
        try:
            shard_items = future.result()
            all_items.extend(shard_items)
            print(f"Shard {shard_number} returned {len(shard_items)} items")
        except Exception as exc:
            print(f"Shard {shard_number} generated an exception: {exc}")

# Process the aggregated results (e.g., sorting, filtering) as needed
# For example, simply printing the count of all retrieved items
print(f"Total items retrieved from all shards: {len(all_items)}")
```

Before running this code, make sure to replace `YourTableName` and `YourGSIName` with the actual table and GSI names from your DynamoDB setup. Also, adjust `total_shards`, `time_start`, and `time_end` variables according to your specific requirements.

This script queries each shard for items within the specified time range and aggregates the results.

Using Global Secondary Indexes to create an eventually consistent replica in DynamoDB

You can use a global secondary index to create an eventually consistent replica of a table. Creating a replica can allow you to do the following:

- **Set different provisioned read capacity for different readers.** For example, suppose that you have two applications: One application handles high-priority queries and needs the highest levels of read performance, whereas the other handles low-priority queries that can tolerate throttling of read activity.

  If both of these applications read from the same table, a heavy read load from the low-priority application could consume all the available read capacity for the table. This would throttle the high-priority application's read activity.

  Instead, you can create a replica through a global secondary index whose read capacity you can set separate from that of the table itself. You can then have your low-priority app query the replica instead of the table.
- **Eliminate reads from a table entirely.** For example, you might have an application that captures a high volume of clickstream activity from a website, and you don't want to risk having reads interfere with that. You can isolate this table and prevent reads by other applications (see Using IAM policy conditions for fine-grained access control), while letting other applications read a replica created using a global secondary index.

To create a replica, set up a global secondary index that has the same key schema as the parent table, with some or all of the non-key attributes projected into it. In applications, you can direct some or all read activity to this global secondary index rather than to the parent table. You can then adjust the provisioned read capacity of the global secondary index to handle those reads without changing the parent table's provisioned read capacity.

There is always a short propagation delay between a write to the parent table and the time when the written data appears in the index. In other words, your applications should take into account that the global secondary index replica is only *eventually consistent* with the parent table.

You can create multiple global secondary index replicas to support different read patterns. When you create the replicas, project only the attributes that each read pattern actually requires. An application can then consume less provisioned read capacity to obtain only the data it needs rather

than having to read the item from the parent table. This optimization can result in significant cost savings over time.

Best practices for storing large items and attributes in DynamoDB

Amazon DynamoDB limits the size of each item that you store in a table to 400 KB (see Item size). If your application needs to store more data in an item than the DynamoDB size limit permits, you can try compressing one or more large attributes or breaking the item into multiple items (efficiently indexed by sort keys). You can also store the item as an object in Amazon Simple Storage Service (Amazon S3) and store the Amazon S3 object identifier in your DynamoDB item.

As a best practice, you should utilize the `ReturnConsumedCapacity` parameter when writing items to monitor and alert on items sizes that approach the 400 KB maximum item size. Exceeding the maximum item size will result in failed write attempts. DynamoDB will return a ValidationException error. Monitoring and alerting on item sizes will enable you to mitigate the items size issues before they impact your application.

Compressing large attribute values

Compressing large attribute values can let them fit within item limits in DynamoDB and reduce your storage costs. Compression algorithms such as GZIP or LZO produce binary output that you can then store in a `Binary` attribute type within the item.

As an example, consider a table that stores messages written by forum users. Such messages often contain long strings of text, which are candidates for compression. While compression can reduce item sizes, the downside is that the compressed attribute values are not useful for filtering.

For sample code that demonstrates how to compress such messages in DynamoDB, see the following:

- Example: Handling binary type attributes using the AWS SDK for Java document API
- Example: Handling binary type attributes using the AWS SDK for .NET low-level API

Vertical partitioning

An alternative solution to dealing with large items is to break them down into smaller chunks of data and associating all relevant items by the partition key value. You can then use a sort key string

to identify the associated information stored alongside it. By doing this, and having multiple items grouped by the same partition key value, you are creating an *item collection*.

For more information on this approach, see:

- Use vertical partitioning to scale data efficiently in Amazon DynamoDB
- Implement vertical partitioning in Amazon DynamoDB using AWS Glue

Storing large attribute values in Amazon S3

As mentioned previously, you can also use Amazon S3 to store large attribute values that cannot fit in a DynamoDB item. You can store them as an object in Amazon S3 and then store the object identifier in your DynamoDB item.

You can also use the object metadata support in Amazon S3 to provide a link back to the parent item in DynamoDB. Store the primary key value of the item as Amazon S3 metadata of the object in Amazon S3. Doing this often helps with maintenance of the Amazon S3 objects.

For example, consider the `ProductCatalog` table. Items in this table store information about item price, description, book authors, and dimensions for other products. If you wanted to store an image of each product that was too large to fit in an item, you could store the images in Amazon S3 instead of in DynamoDB.

When implementing this strategy, keep the following in mind:

- DynamoDB doesn't support transactions that cross Amazon S3 and DynamoDB. Therefore, your application must deal with any failures, which could include cleaning up orphaned Amazon S3 objects.
- Amazon S3 limits the length of object identifiers. So you must organize your data in a way that doesn't generate excessively long object identifiers or violate other Amazon S3 constraints.

For more information about how to use Amazon S3, see the Amazon Simple Storage Service User Guide.

Best practices for handling time series data in DynamoDB

General design principles in Amazon DynamoDB recommend that you keep the number of tables you use to a minimum. For most applications, a single table is all you need. However, for time series data, you can often best handle it by using one table per application per period.

Design pattern for time series data

Consider a typical time series scenario, where you want to track a high volume of events. Your write access pattern is that all the events being recorded have today's date. Your read access pattern might be to read today's events most frequently, yesterday's events much less frequently, and then older events very little at all. One way to handle this is by building the current date and time into the primary key.

The following design pattern often handles this kind of scenario effectively:

- Create one table per period, provisioned with the required read and write capacity and the required indexes.
- Before the end of each period, prebuild the table for the next period. Just as the current period ends, direct event traffic to the new table. You can assign names to these tables that specify the periods they have recorded.
- As soon as a table is no longer being written to, reduce its provisioned write capacity to a lower value (for example, 1 WCU), and provision whatever read capacity is appropriate. Reduce the provisioned read capacity of earlier tables as they age. You might choose to archive or delete the tables whose contents are rarely or never needed.

The idea is to allocate the required resources for the current period that will experience the highest volume of traffic and scale down provisioning for older tables that are not used actively, therefore saving costs. Depending on your business needs, you might consider write sharding to distribute traffic evenly to the logical partition key. For more information, see Using write sharding to distribute workloads evenly in your DynamoDB table.

Time series table examples

The following is a time series data example in which the current table is provisioned at a higher read/write capacity and the older tables are scaled down because they are accessed infrequently.

Current table — Provisioned at: WCU=750 and RCU=300

Primary Key		Attributes		
Partition Key	Sort Key	Radiant Intensity	Wavelength	...
2018-03-15	00:00:00.002	17.372 W/Sr	713 nm	...
2018-03-15	00:00:00.004	17.385 W/Sr	712 nm	...
2018-03-15	00:00:00.005	17.478 W/Sr	708 nm	...
2018-03-15	00:00:00.007	19.172 W/Sr	674 nm	...
. . .	. . .	. . .	. . .	...

Previous table — Provisioned at: WCU=1 and RCU=100

Primary Key		Attributes		
Partition Key	Sort Key	Radiant Intensity	Wavelength	...
2018-03-14	00:00:00.001	16.473 W/Sr	512	...
2018-03-14	00:00:00.003	16.489 W/Sr	519	...
2018-03-14	00:00:00.004	16.814 W/Sr	522	...
2018-03-14	00:00:00.006	16.719 W/Sr	506	...
. . .	. . .	. . .	. . .	...

Older table — Provisioned at: WCU=1 and RCU=1

Primary Key		Attributes		
Partition Key	Sort Key	Radiant Intensity	Wavelength	...
2018-03-10	00:00:00.001	13.669 W/Sr	456	...
2018-03-10	00:00:00.002	13.522 W/Sr	459	...
2018-03-10	00:00:00.004	13.596 W/Sr	457	...
2018-03-10	00:00:00.005	15.721 W/Sr	425	...
. . .	. . .	. . .	. . .	...

Best practices for managing many-to-many relationships in DynamoDB tables

Adjacency lists are a design pattern that is useful for modeling many-to-many relationships in Amazon DynamoDB. More generally, they provide a way to represent graph data (nodes and edges) in DynamoDB.

Adjacency list design pattern

When different entities of an application have a many-to-many relationship between them, the relationship can be modeled as an adjacency list. In this pattern, all top-level entities (synonymous to nodes in the graph model) are represented using the partition key. Any relationships with other entities (edges in a graph) are represented as an item within the partition by setting the value of the sort key to the target entity ID (target node).

The advantages of this pattern include minimal data duplication and simplified query patterns to find all entities (nodes) related to a target entity (having an edge to a target node).

A real-world example where this pattern has been useful is an invoicing system where invoices contain multiple bills. One bill can belong in multiple invoices. The partition key in this example is either an `InvoiceID` or a `BillID`. `BillID` partitions have all attributes specific to bills. `InvoiceID` partitions have an item storing invoice-specific attributes, and an item for each `BillID` that rolls up to the invoice.

The schema looks like the following.

Table	Primary Key		Data Attributes...	
	Partition Key	*Sort Key (and GSI PK)*		
	`Invoice-92551`	**Inv_ID:** Invoice-92551 *(invoice ID)*	**Dated:** 2018-02-07 *(date created)*	More attributes of this invoice...
		Bill_ID: Bill-4224663 *(bill ID)*	**Dated:** 2017-12-03 *(date created)*	Attributes of this bill *in this invoice*..
		Bill_ID: Bill-4224687 *(bill ID)*	**Dated:** 2018-01-09 *(date created)*	Attributes of this bill *in this invoice*..
	`Invoice-92552`	**Inv_ID:** Invoice-92552 *(invoice ID)*	**Dated:** 2018-03-04 *(date created)*	More attributes of this invoice...
		Bill_ID: Bill-4224687 *(bill ID)*	**Dated:** 2018-01-09 *(date created)*	Attributes of this bill *in this invoice*..
	`Bill-4224663`	**Bill_ID:** Bill-4224663 *(bill ID)*	**Dated:** 2017-12-03 *(date created)*	More attributes of this bill...
	`Bill-4224687`	**Bill_ID:** Bill-4224687 *(bill ID)*	**Dated:** 2018-01-09 *(date created)*	More attributes of this bill...

Using the preceding schema, you can see that all bills for an invoice can be queried using the primary key on the table. To look up all invoices that contain a part of a bill, create a global secondary index on the table's sort key.

The projections for the global secondary index look like the following.

GSI	Primary Key *Partition Key*	Projected Attributes...	
	Bill-4224663	**Bill_ID:** Bill-4224663 *(table primary key)*	Attributes of this bill...
		Inv_ID: Invoice-92551 *(table primary key)*	Attributes of this bill *in this invoice*..
	Bill-4224687	**Bill_ID:** Bill-4224687 *(table primary key)*	Attributes of this bill...
		Inv_ID: Invoice-92551 *(table primary key)*	Attributes of this bill *in this invoice*..
		Inv_ID: Invoice-92552 *(table primary key)*	Attributes of this bill *in this invoice*..
	Invoice-92551	**Inv_ID:** Invoice-92551 *(table primary key)*	Attributes of this invoice...
	Invoice-92552	**Inv_ID:** Invoice-92552 *(table primary key)*	Attributes of this invoice...

Materialized graph pattern

Many applications are built around understanding rankings across peers, common relationships between entities, neighbor entity state, and other types of graph style workflows. For these types of applications, consider the following schema design pattern.

TABLE	Primary Key		Attributes		
	PK (NodeId)	SK (TypeTarget, GSI 2 SK)	Data	GSI PK	Graph Projections
	1	DATE|2|BIRTH	1980-12-19	Hash(Person.Data)	...
		PERSON|1	Data (GSI1 SK)	GSI PK	
			John Doe	Hash(Person.Data)	
		PERSON|5|FRIEND	Data	GSI PK	
			Jane Smith	Hash(Person.Data)	
		PLACE|4|BIRTH	Data	GSI PK	
			USA|Texas|Austin	Hash(Person.Data)	
		SKILL|6	Data	GSI PK	
			Java Developer|Senior	Hash(Person.Data)	
	2	DATE|2	Data	GSI PK	
			1980-12-19	0	
	3	PLACE|3	Data	GSI PK	
			UK|England|London	0	
	4	PLACE|4	Data	GSI PK	
			USA|Texas|Austin	0	
	5	DATE|2|BIRTH	Data	GSI PK	
			1980-12-19	Hash(Person.Data)	
		PERSON|5	Data	GSI PK	
			Jane Smith	Hash(Person.Data)	
		PERSON|1|FRIEND	Data	GSI PK	
			John Doe	Hash(Person.Data)	
		PLACE|3|BIRTH	Data	GSI PK	
			UK|England|London	Hash(Person.Data)	
		SKILL|7	Data	GSI PK	
			Guitar|Advanced	Hash(Person.Data)	
	6	SKILL|6	Data	GSI PK	
			Java Developer	0	
	7	SKILL|7	Data	GSI PK	
			Guitar	0	

GSI 1	Primary Key		Attributes		
	GSI PK	GSI 1 SK (Data)	NodeID	TypeTarget	Graph Projections
	O-N	1980-12-19	2	DATE|2	...
			NodeID	TypeTarget	
			1	DATE|2|BIRTH	
			NodeID		
			5		
		Guitar	NodeID	TypeTarget	
			7	SKILL|7	
		Guitar|Advanced	NodeID		
			5		
		Jane Smith	NodeID	TypeTarget	
			5	Person|5	
			NodeID	TypeTarget	
			1	Person|5|FRIEND	
		Java Developer	NodeID	TypeTarget	
			6	SKILL|6	
		Java Developer|Senior	NodeID		
			1		
		John Doe	NodeID	TypeTarget	
			1	Person|1	
			NodeID	TypeTarget	
			5	Person|1|FRIEND	
		UK|England|London	NodeID	TypeTarget	
			3	PLACE|3	
			NodeID	TypeTarget	
			1	PLACE|3|BIRTH	
		USA|Texas|Austin	NodeID	TypeTarget	
			4	PLACE|4	
			NodeID	TypeTarget	
			5	PLACE|4|BIRTH	

	Primary Key		Attributes		
	GSI PK	GSI 2 SK (TypeTarget)			
GSI 2	O-N	DATE|2	NodeID	Data	Graph Projections
			2	1980-12-19	...
		DATE|2|BIRTH	NodeID		
			1		
			NodeID		
			5		
		PERSON|1	NodeID	Data	
			1	John Doe	
		PERSON|1|FRIEND	NodeID		
			5		
		PERSON|5	NodeID	Data	
			5	Jane Smith	
		PERSON|5|FRIEND	NodeID		
			1		
		PLACE|3	NodeID	Data	
			3	UK|England|London	
		PLACE|3|BIRTH	NodeID		
			5		
		PLACE|4	NodeID	Data	
			4	USA|texas|Austin	
		PLACE|4|BIRTH	NodeID		
			1		
		SKILL|6	NodeID	Data	
			6	Java Developer	
			NodeID	Data	
			1	Java Developer|Senior	
		SKILL|7	NodeID	Data	
			7	Guitar	
			NodeID	Data	
			5	Guitar|Advanced	

The preceding schema shows a graph data structure that is defined by a set of data partitions containing the items that define the edges and nodes of the graph. Edge items contain a `Target` and a `Type` attribute. These attributes are used as part of a composite key name "TypeTarget" to identify the item in a partition in the primary table or in a second global secondary index.

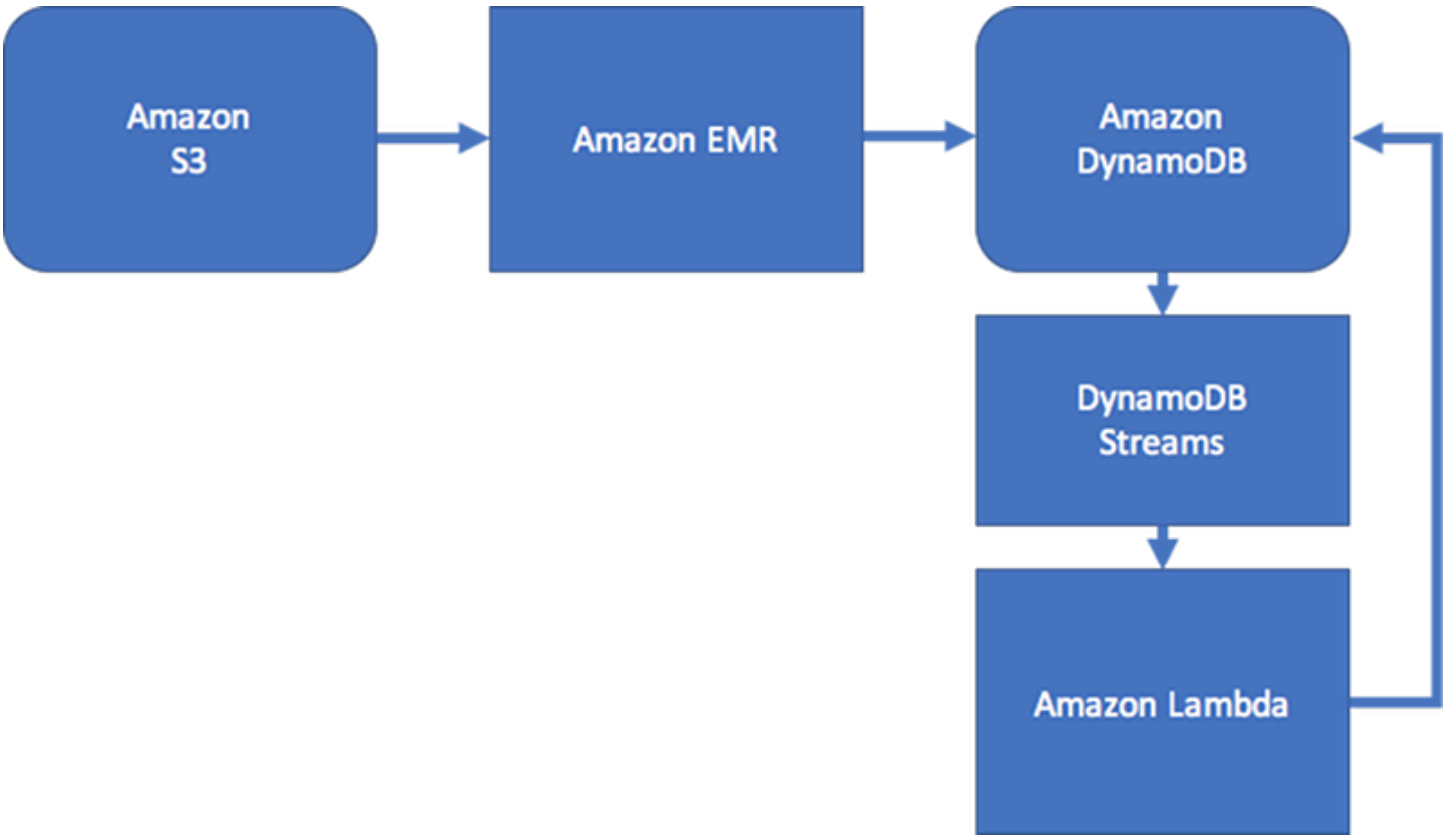
The first global secondary index is built on the `Data` attribute. This attribute uses global secondary index-overloading as described earlier to index several different attribute types, namely `Dates`, `Names`, `Places`, and `Skills`. Here, one global secondary index is effectively indexing four different attributes.

As you insert items into the table, you can use an intelligent sharding strategy to distribute item sets with large aggregations (birthdate, skill) across as many logical partitions on the global secondary indexes as are needed to avoid hot read/write problems.

The result of this combination of design patterns is a solid datastore for highly efficient real-time graph workflows. These workflows can provide high-performance neighbor entity state and edge aggregation queries for recommendation engines, social-networking applications, node rankings, subtree aggregations, and other common graph use cases.

If your use case isn't sensitive to real-time data consistency, you can use a scheduled Amazon EMR process to populate edges with relevant graph summary aggregations for your workflows. If your application doesn't need to know immediately when an edge is added to the graph, you can use a scheduled process to aggregate results.

To maintain some level of consistency, the design could include Amazon DynamoDB Streams and AWS Lambda to process edge updates. It could also use an Amazon EMR job to validate results on a regular interval. This approach is illustrated by the following diagram. It is commonly used in social networking applications, where the cost of a real-time query is high and the need to immediately know individual user updates is low.



IT service-management (ITSM) and security applications generally need to respond in real time to entity state changes composed of complex edge aggregations. Such applications need a system that can support real-time multiple node aggregations of second- and third-level relationships, or complex edge traversals. If your use case requires these types of real-time graph query workflows, we recommend that you consider using Amazon Neptune to manage these workflows.

> **Note**
>
> If you need to query highly connected datasets or execute queries that need to traverse multiple nodes (also known as multi-hop queries) with millisecond latency, you should consider using Amazon Neptune. Amazon Neptune is a purpose-built, high-performance graph database engine optimized for storing billions of relationships and querying the graph with millisecond latency.

Best practices for querying and scanning data in DynamoDB

This section covers some best practices for using `Query` and `Scan` operations in Amazon DynamoDB.

Performance considerations for scans

In general, `Scan` operations are less efficient than other operations in DynamoDB. A `Scan` operation always scans the entire table or secondary index. It then filters out values to provide the result you want, essentially adding the extra step of removing data from the result set.

If possible, you should avoid using a `Scan` operation on a large table or index with a filter that removes many results. Also, as a table or index grows, the `Scan` operation slows. The `Scan` operation examines every item for the requested values and can use up the provisioned throughput for a large table or index in a single operation. For faster response times, design your tables and indexes so that your applications can use `Query` instead of `Scan`. (For tables, you can also consider using the `GetItem` and `BatchGetItem` APIs.)

Alternatively, you can design your application to use `Scan` operations in a way that minimizes the impact on your request rate. This can include modeling when it might be more efficient to use a global secondary index instead of a `Scan` operation. Further information on this process is in the following video.

Modeling low velocity access patterns

Avoiding sudden spikes in read activity

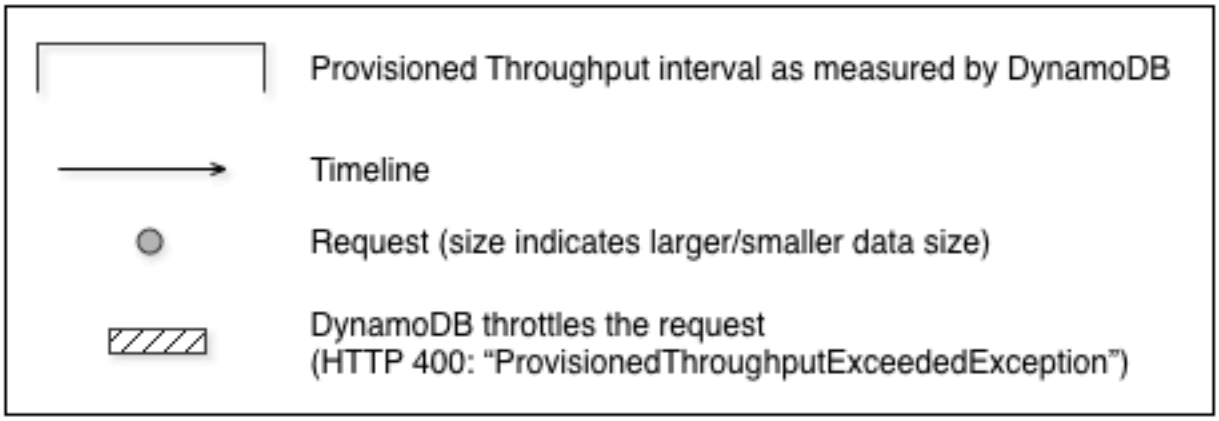
When you create a table, you set its read and write capacity unit requirements. For reads, the capacity units are expressed as the number of strongly consistent 4 KB data read requests per

second. For eventually consistent reads, a read capacity unit is two 4 KB read requests per second. A `Scan` operation performs eventually consistent reads by default, and it can return up to 1 MB (one page) of data. Therefore, a single `Scan` request can consume (1 MB page size / 4 KB item size) / 2 (eventually consistent reads) = 128 read operations. If you request strongly consistent reads instead, the `Scan` operation would consume twice as much provisioned throughput—256 read operations.

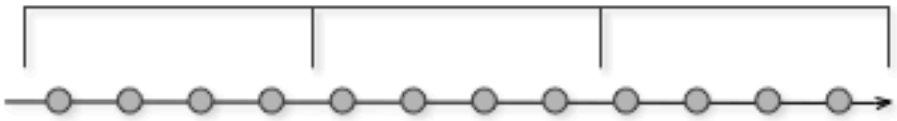
This represents a sudden spike in usage, compared to the configured read capacity for the table. This usage of capacity units by a scan prevents other potentially more important requests for the same table from using the available capacity units. As a result, you likely get a `ProvisionedThroughputExceeded` exception for those requests.

The problem is not just the sudden increase in capacity units that the `Scan` uses. The scan is also likely to consume all of its capacity units from the same partition because the scan requests read items that are next to each other on the partition. This means that the request is hitting the same partition, causing all of its capacity units to be consumed, and throttling other requests to that partition. If the request to read data is spread across multiple partitions, the operation would not throttle a specific partition.

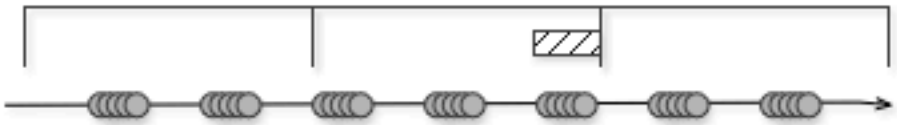
The following diagram illustrates the impact of a sudden spike of capacity unit usage by `Query` and `Scan` operations, and its impact on your other requests against the same table.



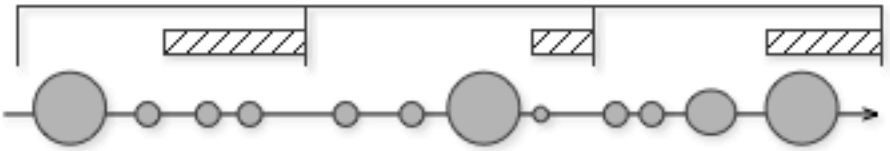
1. Good: Even distribution of requests and size



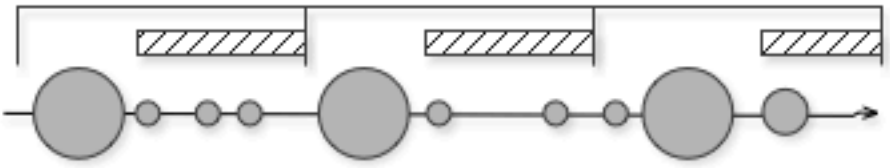
2. Not as Good: Frequent requests in bursts



3. Bad: A few random large requests



4. Bad: Large scan operations



As illustrated here, the usage spike can impact the table's provisioned throughput in several ways:

1. Good: Even distribution of requests and size

2. Not as good: Frequent requests in bursts
3. Bad: A few random large requests
4. Bad: Large scan operations

Instead of using a large `Scan` operation, you can use the following techniques to minimize the impact of a scan on a table's provisioned throughput.

- **Reduce page size**

  Because a Scan operation reads an entire page (by default, 1 MB), you can reduce the impact of the scan operation by setting a smaller page size. The `Scan` operation provides a *Limit* parameter that you can use to set the page size for your request. Each `Query` or `Scan` request that has a smaller page size uses fewer read operations and creates a "pause" between each request. For example, suppose that each item is 4 KB and you set the page size to 40 items. A `Query` request would then consume only 20 eventually consistent read operations or 40 strongly consistent read operations. A larger number of smaller `Query` or `Scan` operations would allow your other critical requests to succeed without throttling.
- **Isolate scan operations**

  DynamoDB is designed for easy scalability. As a result, an application can create tables for distinct purposes, possibly even duplicating content across several tables. You want to perform scans on a table that is not taking "mission-critical" traffic. Some applications handle this load by rotating traffic hourly between two tables—one for critical traffic, and one for bookkeeping. Other applications can do this by performing every write on two tables: a "mission-critical" table, and a "shadow" table.

Configure your application to retry any request that receives a response code that indicates you have exceeded your provisioned throughput. Or, increase the provisioned throughput for your table using the `UpdateTable` operation. If you have temporary spikes in your workload that cause your throughput to exceed, occasionally, beyond the provisioned level, retry the request with exponential backoff. For more information about implementing exponential backoff, see Error retries and exponential backoff.

Taking advantage of parallel scans

Many applications can benefit from using parallel `Scan` operations rather than sequential scans. For example, an application that processes a large table of historical data can perform a parallel

scan much faster than a sequential one. Multiple worker threads in a background "sweeper" process could scan a table at a low priority without affecting production traffic. In each of these examples, a parallel `Scan` is used in such a way that it does not starve other applications of provisioned throughput resources.

Although parallel scans can be beneficial, they can place a heavy demand on provisioned throughput. With a parallel scan, your application has multiple workers that are all running `Scan` operations concurrently. This can quickly consume all of your table's provisioned read capacity. In that case, other applications that need to access the table might be throttled.

A parallel scan can be the right choice if the following conditions are met:

- The table size is 20 GB or larger.
- The table's provisioned read throughput is not being fully used.
- Sequential `Scan` operations are too slow.

Choosing TotalSegments

The best setting for `TotalSegments` depends on your specific data, the table's provisioned throughput settings, and your performance requirements. You might need to experiment to get it right. We recommend that you begin with a simple ratio, such as one segment per 2 GB of data. For example, for a 30 GB table, you could set `TotalSegments` to 15 (30 GB / 2 GB). Your application would then use 15 workers, with each worker scanning a different segment.

You can also choose a value for `TotalSegments` that is based on client resources. You can set `TotalSegments` to any number from 1 to 1000000, and DynamoDB lets you scan that number of segments. For example, if your client limits the number of threads that can run concurrently, you can gradually increase `TotalSegments` until you get the best `Scan` performance with your application.

Monitor your parallel scans to optimize your provisioned throughput use, while also making sure that your other applications aren't starved of resources. Increase the value for `TotalSegments` if you don't consume all of your provisioned throughput but still experience throttling in your `Scan` requests. Reduce the value for `TotalSegments` if the `Scan` requests consume more provisioned throughput than you want to use.

Best practices for DynamoDB table design

General design principles in Amazon DynamoDB recommend that you keep the number of tables you use to a minimum. In the majority of cases, we recommend that you consider using a single table. However if a single or small number of tables are not viable, these guidelines may be of use.

- The per account limit cannot be increased above 10,000 tables per account. If your application requires more tables, plan for distributing the tables across multiple accounts. For more information see service, account, and table quotas in Amazon DynamoDB.
- Consider control plane limits for concurrent control plane operations that might impact your table management.
- Work with AWS solution architects to validate your design patterns for multi-tenant designs.

Using DynamoDB global tables

Global tables build on Amazon DynamoDB's global footprint to provide you with a fully managed, multi-Region, and multi-active database that can deliver fast and local, read and write performance for massively scaled, global applications. Global tables replicate your DynamoDB tables automatically across your choice of AWS Regions. No application changes are required because global tables use existing DynamoDB APIs. There are no upfront costs or commitments for using global tables, and you pay only for the resources you use.

This guide explains how to use DynamoDB global tables effectively. It provides key facts about global tables, explains the feature's primary use cases, describes the two consistency modes, introduces a taxonomy of three different write models you should consider, walks through the four main request routing choices you might implement, discusses ways to evacuate a Region that's live or a Region that's offline, explains how to think about throughput capacity planning, and provides a checklist of things to consider when you deploy global tables.

This guide fits into a larger context of AWS multi-Region deployments, as covered in the AWS Multi-Region Fundamentals whitepaper and the Data resiliency design patterns with AWS video.

Topics

- Key facts about DynamoDB global table design
- Key facts about MREC
- Key facts about MRSC

- MREC DynamoDB global table use cases
- Write modes with DynamoDB global tables
- Routing strategies in DynamoDB
- Evacuation processes
- Throughput capacity planning for DynamoDB global tables
- Preparation checklist for DynamoDB global tables
- Conclusion and resources

Key facts about DynamoDB global table design

- There are two versions of global tables: the current version Global Tables version 2019.11.21 (Current) (sometimes called "V2"), and Global tables version 2017.11.29 (Legacy) (sometimes called "V1"). This guide focuses exclusively on the current version.
- DynamoDB (without global tables) is a Regional service, which means that it is highly available and intrinsically resilient to failures of infrastructure, including the failure of an entire Availability Zone. A single-Region DynamoDB table is designed for 99.99% availability. For more information, see the DynamoDB service-level agreement (SLA).
- A DynamoDB global table replicates its data between two or more Regions. A multi-Region DynamoDB table is designed for 99.999% availability. With proper planning, global tables can help create an architecture that is resilient to Regional failures.
- DynamoDB doesn't have a global endpoint. All requests are made to a Regional endpoint that accesses the global table instance that's local to that Region.
- Calls to DynamoDB should not go across Regions. The best practice is for an application that is homed to one Region to directly access only the local DynamoDB endpoint for its Region. If problems are detected within a Region (in the DynamoDB layer or in the surrounding stack), end user traffic should be routed to a different application endpoint that's hosted in a different Region. Global tables ensure that the application homed in every Region has access to the same data.

Consistency modes

When you create a global table, you configure its consistency mode. Global tables support two consistency modes: multi-Region eventual consistency (MREC) and multi-Region strong consistency (MRSC) which was introduced in June 2025.

If you don't specify a consistency mode when you create a global table, the global table defaults to MREC. A global table can't contain replicas that are configured with different consistency modes. You can't change a global table's consistency mode after its creation.

Key facts about MREC

- Global tables that use MRSC also employ an active-active replication model. From the perspective of DynamoDB, the table in each Region has equal standing to accept read and write requests. After receiving a write request, the local replica table replicates the write operation to other participating remote Regions in the background.
- Items are replicated individually. Items that are updated within a single transaction might not be replicated together.
- Each table partition in the source Region replicates its write operations in parallel with every other partition. The sequence of write operations within a remote Region might not match the sequence of write operations that happened within the source Region. For more information about table partitions, see the blog post Scaling DynamoDB: How partitions, hot keys, and split for heat impact performance.
- A newly written item is usually propagated to all replica tables within a second. Nearby Regions tend to propagate faster.
- Amazon CloudWatch provides a `ReplicationLatency` metric for each Region pair. It is calculated by looking at arriving items, comparing their arrival time with their initial write time, and computing an average. Timings are stored within CloudWatch in the source Region. Viewing the average and maximum timings can be useful for determining the average and worst-case replication lag. There is no SLA on this latency.
- If an individual item is updated at about the same time (within this `ReplicationLatency` window) in two different Regions, and the second write operation happens before the first write operation was replicated, there's a potential for write conflicts. Global tables that use MREC resolve such conflicts by using a last writer wins mechanism, based on the timestamp of the write operations. The first operation "loses" to the second operation. These conflicts aren't recorded in CloudWatch or AWS CloudTrail.
- Each item has a last write timestamp held as a private system property. The last writer wins approach is implemented by using a conditional write operation that requires the incoming item's timestamp to be greater than the existing item's timestamp.
- A global table replicates all items to all participating Regions. If you want to have different replication scopes, you can create multiple global tables and assign each table different participating Regions.

- The local Region accepts write operations even if the replica Region is offline or `ReplicationLatency` grows. The local table continues to attempt replicating items to the remote table until each item succeeds.
- In the unlikely event that a Region goes fully offline, when it comes back online later, all pending outbound and inbound replications will be retried. No special action is required to bring the tables back in sync. The *last writer wins* mechanism ensures that the data eventually becomes consistent.
- You can add a new Region to a DynamoDB MREC table at any time. DynamoDB handles the initial sync and ongoing replication. You can also remove a Region (even the original Region), and this will delete the local table in that Region.

Key facts about MRSC

- Global tables that use MRSC also employ an active-active replication model. From the perspective of DynamoDB, the table in each Region has equal standing to accept read and write requests. Item changes in an MRSC global table replica are **synchronously** replicated to at least one other Region before the write operation returns a successful response.
- Strongly consistent read operations on any MRSC replica always return the latest version of an item. Conditional write operations always evaluate the condition expression against the latest version of an item. Updates always operate against the latest version of an item.
- Eventually consistent read operations on an MRSC replica might not include changes that recently occurred in another Region, and might not even include changes that very recently occurred in the same Region.
- A write operation fails with a `ReplicatedWriteConflictException` exception when it attempts to modify an item that is already being modified in another Region. Write operations that fail with the `ReplicatedWriteConflictException` exception can be retried and will succeed if the item is no longer being modified in another Region.
- With MRSC, latencies are higher for write operations and for strongly consistent read operations. These operations require cross-Region communication. This communication can add latency that increases based on the round-trip latency between the Region being accessed and the nearest Region participating in the global table. For more information, see the AWS re:Invent 2024 presentation, Multi-Region strong consistency with DynamoDB global tables. Eventually consistent read operations experience no extra latency. There is an open source tester tool that lets you experimentally calculate these latencies with your Regions.
- Items are replicated individually. Global tables using MRSC do not support the transaction APIs.

- A MRSC global table must be deployed in exactly three Regions. You can configure a MRSC global table with three replicas, or with two replicas and one witness. A witness is a component of an MRSC global table that contains recent data written to global table replicas. A witness provides an optional alternative to a full replica while supporting MRSC's availability architecture. You can't perform read or write operations on a witness. A witness doesn't incur storage or write costs. A witness is located within a different Region from the two replicas.
- To create an MRSC global table, you add one replica and a witness, or add two replicas to an existing DynamoDB table that contains no data. You cannot add additional replicas to an existing MRSC global table. You can't delete a single replica or a witness from an MRSC global table. You can delete two replicas, or delete one replica and a witness, from an MRSC global table. The second scenario converts the remaining replica to a single-Region DynamoDB table.
- You can determine whether an MRSC global table has a witness configured, and which Region in which it's configured, from the output of the DescribeTable API. The witness is owned and managed by DynamoDB and doesn't appear in your AWS account in the Region where it's configured.
- MRSC global tables are available in the following Region sets:
  - US Region set: US East (N. Virginia), US East (Ohio), US West (Oregon)
  - EU Region set: Europe (Ireland), Europe (London), Europe (Paris), Europe (Frankfurt)
  - AP Region set: Asia Pacific (Tokyo), Asia Pacific (Seoul), and Asia Pacific (Osaka)
- MRSC global tables can't span Region sets. For example, an MRSC global table can't contain replicas from both US and EU Region sets.
- Time to Live (TTL) isn't supported for MRSC global tables.
- Local secondary indexes (LSIs) aren't supported for MRSC global tables.
- CloudWatch Contributor Insights information is only reported for the Region in which an operation occurred.
- The local Region accepts all read and write operations as long as there is a second Region that hosts a replica or witness to establish quorum. If a second Region isn't available, the local Region can only service eventually consistent reads.
- In the unlikely event that a Region goes fully offline, when it comes back online later, it will automatically catch up. Until it's caught up, write operations and strongly consistent read operations *only* to the catching up Region will return errors while requests to other Regions will continue to perform normally. Eventually consistent read operations to the catching up Region will return the data that has so far been propagated into the Region, with usual local consistency

behavior between the leader node and local replicas. No special action is required to bring the tables back in sync.

MREC DynamoDB global table use cases

MREC global tables provides these benefits:

- **Lower-latency read operations.** Place a copy of the data closer to the end user to reduce network latency during read operations. The data is kept as fresh as the `ReplicationLatency` value.
- **Lower-latency write operations.** You can write to a nearby region to reduce network latency and the time taken to achieve the write. The write traffic must be carefully routed to ensure no conflicts. Techniques for routing are discussed in more detail in Routing strategies in DynamoDB.
- **Seamless Region migration.** You can add a new Region and delete the old Region to migrate a deployment from one Region to another without downtime at the data layer.

MREC and MRSC global tables both provide this benefit:

- **Increased resiliency and disaster recovery.** If a Region has degraded performance or a full outage, you can evacuate it. To evacuate means moving away some or all requests going to that Region. Using global tables increases the DynamoDB SLA for monthly uptime percentage from 99.99% to 99.999%. Using MREC supports a recovery point objective (RPO) and recovery time objective (RTO) measured in seconds. Using MRSC supports an RPO of zero.

  For example, Fidelity Investments presented at re:Invent 2022 on how they use DynamoDB global tables for their order management system. Their goal was to achieve reliably low latency processing at a scale they couldn't achieve with on-premises processing while also maintaining resilience to Availability Zone and Regional failures.

If your goal is resiliency and disaster recovery, MRSC tables have higher write latencies and higher strongly consistent read latencies, but support an RPO of zero. MREC global tables support an RPO equal to the replication delay between replicas, usually a few seconds depending on the replica Regions.

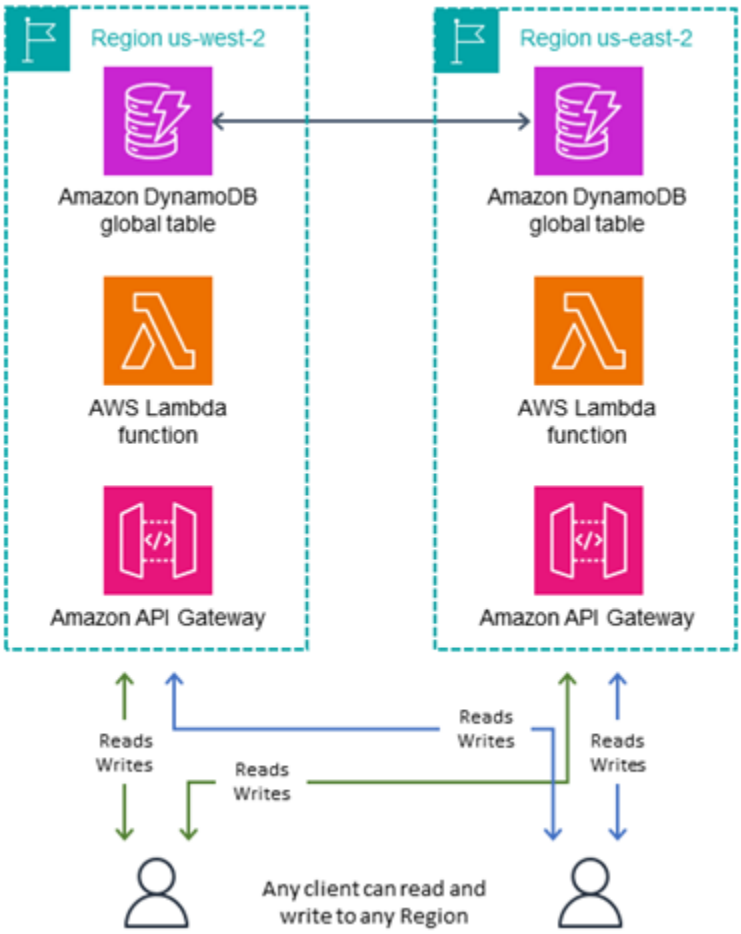
Write modes with DynamoDB global tables

Global tables are always active-active at the table level. However, especially for MREC tables, you might want to treat them as active-passive by controlling how you route write requests. For example, you might decide to route write requests to a single Region to avoid potential write conflicts that can happen with MREC tables.

There are three main managed write patterns, as explained in the next three sections. You should consider which write pattern fits your use case. This choice affects how you route requests, evacuate a Region, and handle disaster recovery. The guidance in later sections depends on your application's write mode.

Write to any Region mode (no primary)

The *write to any Region* mode, illustrated in the following diagram, is fully active-active and doesn't impose restrictions on where a write may occur. Any Region may accept a write at any time. This is the simplest mode, but it can only be used with some types of applications. This mode is suitable for all MRSC tables. It's also suitable for MREC tables when all writers are idempotent, and therefore safely repeatable so that concurrent or repeated write operations across Regions are not in conflict. For example, when a user updates their contact data. This mode also works well for a special case of being idempotent, an append-only dataset where all writes are unique inserts under a deterministic primary key. Lastly, this mode is suitable for MREC where the risk of conflicting writes would be acceptable.



The *write to any Region* mode is the most straightforward architecture to implement. Routing is easier because any Region can be the write target at any time. Failover is easier, because with MRSC tables, the items are always synchronized, and with MRSC tables, any recent writes can be replayed any number of times to any secondary Region. Where possible, you should design for this write mode.

For example, several video streaming services use global tables for tracking bookmarks, reviews, watch status flags, and so on. These deployments use MREC tables because they need replicas scattered around the world, with each replica providing low-latency read and write operations. These deployments can use the *write to any Region* mode as long as they ensure that every write operation is idempotent. This will be the case if every update—for example, setting a new latest time code, assigning a new review, or setting a new watch status—assigns the user's new state directly, and the next correct value for an item doesn't depend on its current value. If, by chance, the user's write requests are routed to different Regions, the last write operation will persist and the global state will settle according to the last assignment. Read operations in this mode will eventually become consistent, delayed by the latest `ReplicationLatency` value.

In another example, a financial services firm uses global tables as part of a system to maintain a running tally of debit card purchases for each customer, to calculate that customer's cash-back rewards. They want to keep a `RunningBalance` item per customer. This write mode is not naturally idempotent because as transactions stream in, they modify the balance by using an `ADD` expression where the new correct value depends on the current value. By using MRSC tables they can still *write to any Region*, because every `ADD` call always operates against the very latest value of the item.

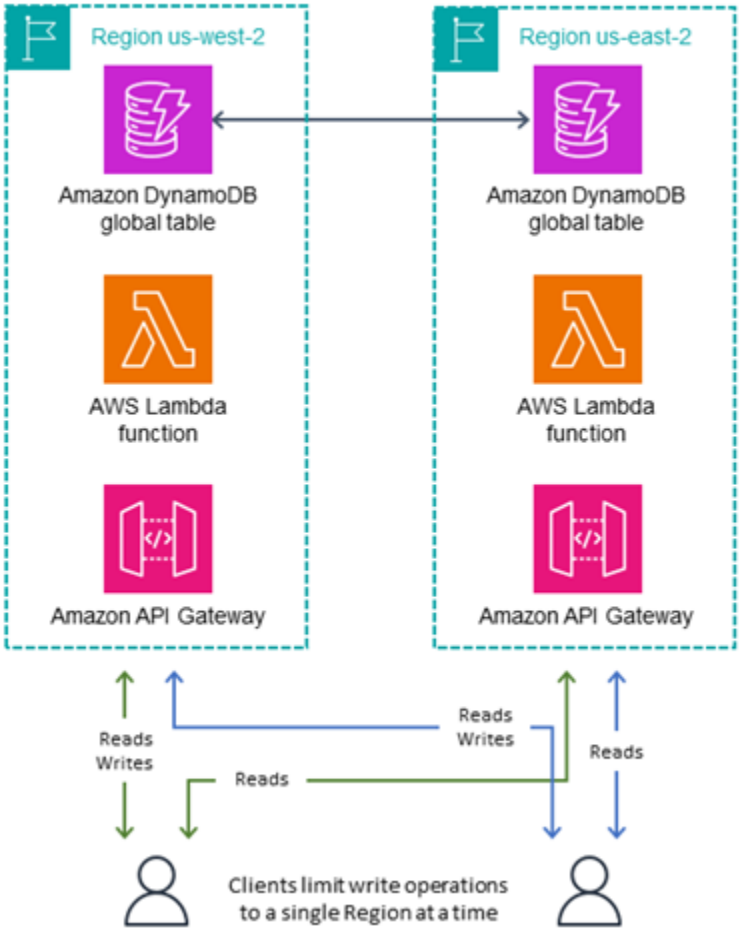
A third example involves a company that provides online ad placement services. This company decided that a low risk of data loss would be acceptable to achieve the design simplifications of the *write to any Region* mode. When they serve ads, they have just a few milliseconds to retrieve enough metadata to determine which ad to show, and then to record the ad impression so they don't repeat the same ad soon. They use global tables to get both low-latency read operations for end users across the world and low-latency write operations. They record all ad impressions for a user within a single item, which is represented as a growing list. They use one item instead of appending to an item collection, so they can remove older ad impressions as part of each write operation without paying for a delete operation. This write operation is not idempotent; if the same end user sees ads served out of multiple Regions at approximately the same time, there's a chance that one write operation for an ad impression could overwrite another. The risk is that a user might see an ad repeated once in a while. They decided that this is acceptable.

Write to one Region (single primary)

The *write to one Region* mode, illustrated in the following diagram, is active-passive and routes all table writes to a single active region. Note that DynamoDB doesn't have a notion of a single active region; the application routing outside DynamoDB manages this. The *write to one Region* mode works well for MREC tables that need to avoid write conflicts by ensuring that write operations flow only to one Region at a time. This write mode helps when you want to use conditional expressions and can't use MRSC for some reason, or when you need to perform transactions. These expressions aren't possible unless you know that you're acting against the latest data, so they require sending all write requests to a single Region that has the latest data.

When you use an MRSC table, you might choose to generally write to one Region for convenience. For example, this can help minimize your infrastructure build-out beyond DynamoDB. The write mode would still be write to any Region because with MRSC you could safely write to any Region at any time without concern of conflict resolution that would cause MREC tables to choose to *write to one Region*.

Eventually consistent reads can go to any replica Regions to achieve lower latencies. Strongly consistent reads must go to the single primary Region.



It's sometimes necessary to change the active Region in response to a Regional failure. Some users change the currently active Region on a regular schedule, such as implementing a follow-the-sun deployment. This places the active Region near the geography that has the most activity (usually where it's daytime, thus the name), which results in the lowest latency read and write operations. It also has the side benefit of calling the Region-changing code daily and making sure that it's well tested before any disaster recovery.

The passive Region(s) may keep a downscaled set of infrastructure surrounding DynamoDB that gets built up only if it becomes the active Region. This guide doesn't cover pilot light and warm standby designs. For a more information, see Disaster Recovery (DR) Architecture on AWS, Part III: Pilot Light and Warm Standby.

Using the *write to one Region* mode works well when you use global tables for low-latency globally distributed read operations. An example is a large social media company that needs to have the

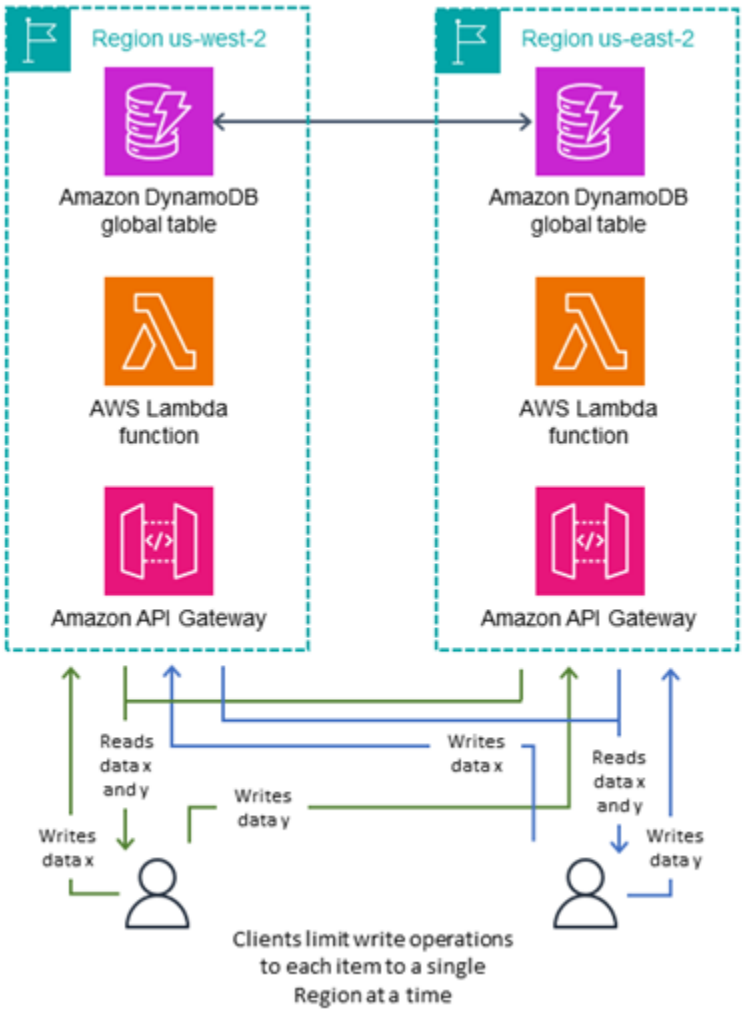
same reference data available in every Region around the world. They don't update the data often, but when they do, they write to only one Region to avoid any potential write conflicts. Read operations are always allowed from any Region.

As another example, consider the financial services company discussed earlier that implemented the daily cash-back calculation. They used *write to any Region* mode to calculate the balance but *write to one Region* mode to track payments. This work requires transactions, which aren't supported in MRSC tables, so it works better with a separate MREC table and *write to one Region* mode.

Write to your Region (mixed primary)

The *write to your Region* write mode, illustrated in the following diagram, works with MREC tables. It assigns different data subsets to different home Regions and allows write operations to an item only through its home Region. This mode is active-passive but assigns the active Region based on the item. Every Region is primary for its own non-overlapping dataset, and write operations must be guarded to ensure proper locality.

This mode is similar to *write to one Region* except that it enables lower-latency write operations, because the data associated with each user can be placed in closer network proximity to that user. It also spreads the surrounding infrastructure more evenly between Regions and requires less work to build out infrastructure during a failover scenario, because all Regions have a portion of their infrastructure already active.



You can determine the home Region for items in several ways:

- **Intrinsic:** Some aspect of the data, such as a special attribute or a value embedded within its partition key, makes its home Region clear. This technique is described in the blog post Use Region pinning to set a home Region for items in an Amazon DynamoDB global table.
- **Negotiated:** The home Region of each dataset is negotiated in some external manner, such as with a separate global service that maintains assignments. The assignment may have a finite duration after which it's subject to renegotiation.
- **Table-oriented:** Instead of creating a single replicating global table, you create the same number of global tables as replicating Regions. Each table's name indicates its home Region. In standard operations, all data is written to the home Region while other Regions keep a read-only copy. During a failover, another Region temporarily adopts write duties for that table.

For example, imagine that you're working for a gaming company. You need low-latency read and write operations for all gamers around the world. You assign each gamer to the Region that's closest to them. That Region takes all their read and write operations, ensuring strong read-after-write consistency. However, when a gamer travels or if their home Region suffers an outage, a complete copy of their data is available in alternative Regions, and the gamer can be assigned to a different home Region.

As another example, imagine that you're working at a video conferencing company. Each conference call's metadata is assigned to a particular Region. Callers can use the Region that's closest to them for lowest latency. If there's a Region outage, using global tables allows quick recovery because the system can move the processing of the call to a different Region where a replicated copy of the data already exists.

To summarize

- Write to any Region mode is suitable for MRSC tables and idempotent calls to MREC tables.
- Write to one Region mode is suitable for non-idempotent calls to MREC tables.
- Write to your Region mode is suitable for non-idempotent calls to MREC tables, where it's important to have clients write to a Region that's close to them.

Routing strategies in DynamoDB

Perhaps the most complex piece of a global table deployment is managing request routing. Requests must first go from an end user to a Region that's chosen and routed in some manner. The request encounters some stack of services in that Region, including a compute layer that perhaps consists of a load balancer backed by an AWS Lambdafunction, container, or Amazon Elastic Compute Cloud (Amazon EC2) node, and possibly other services including another database. That compute layer communicates with DynamoDB It should do that by using the local endpoint for that Region. The data in the global table replicates to all other participating Regions, and each Region has a similar stack of services around its DynamoDB table.

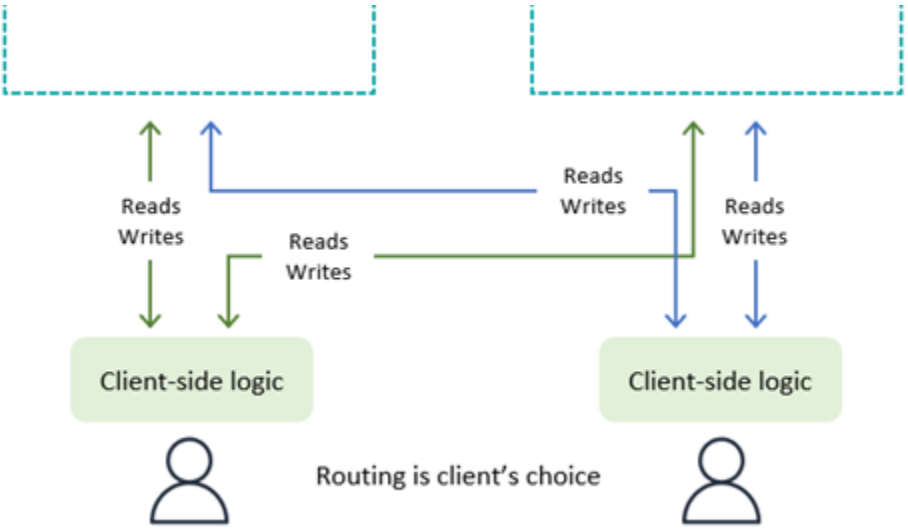
The global table provides each stack in the various Regions with a local copy of the same data. You might consider designing for a single stack in a single Region and anticipate making remote calls to a secondary Region's DynamoDB endpoint if there's an issue with the local DynamoDB table. This is not best practice. If there's an issue in one Region that's caused by DynamoDB (or, more likely, caused by something else in the stack or by another service that depends on DynamoDB), it's best to route the end user to another Region for processing and use that other Region's compute

layer, which will talk to its local DynamoDB endpoint. This approach routes around the problematic Region entirely. To ensure resiliency, you need replication across multiple Regions: replication of the compute layer as well as the data layer.

There are numerous alternative techniques to route an end user request to a Region for processing. The optimum choice depends on your write mode and your failover considerations. This section discusses four options: client-driven, compute-layer, Route 53, and Global Accelerator.

Client-driven request routing

With client-driven request routing, illustrated in the following diagram, the end user client (an application, a web page with JavaScript, or another client) keeps track of the valid application endpoints (for example, an Amazon API Gateway endpoint rather than a literal DynamoDB endpoint) and uses its own embedded logic to choose the Region to communicate with. It might choose based on random selection, lowest observed latencies, highest observed bandwidth measurements, or locally performed health checks.



As an advantage, client-driven request routing can adapt to things such as real-world public internet traffic conditions to switch Regions if it notices any degraded performance. The client must be aware of all potential endpoints, but launching a new Regional endpoint is not a frequent occurrence.

With *write to any Region* mode, a client can unilaterally select its preferred endpoint. If its access to one Region becomes impaired, the client can route to another endpoint.

With the *write to one Region* mode, the client will need a mechanism to route its writes to the currently active region. This could be as basic as empirically testing which region is presently

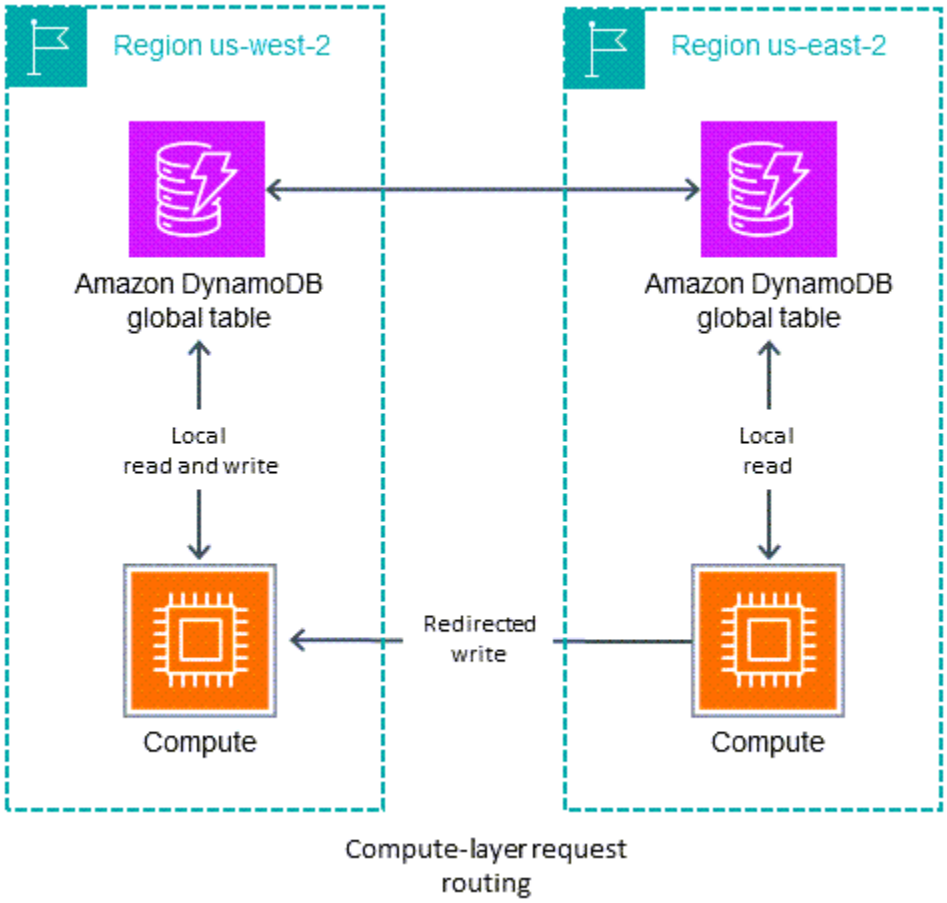
accepting writes (noting any write rejections and falling back to an alternate) or as complex as calling a global coordinator to query for the current application state (perhaps built on the Amazon Application Recovery Controller (ARC) (ARC) routing control which provides a 5-region quorum-driven system to maintain global state for needs such as this). The client can decide if reads can go to any Region for eventual consistency or must be routed to the active region for strong consistency. For further information see How Route 53 works.

With the *write to your Region* mode, the client needs to determine the home region for the data set it's working against. For example, if the client corresponds to a user account and each user account is homed to a Region, the client can request the appropriate endpoint from a global login system.

For example, a financial services company that helps users manage their business finances via the web could use global tables with a *write to your Region* mode. Each user must login to a central service. That service returns credentials and the endpoint for the Region where those credentials will work. The credentials are valid for a short time. After that the webpage auto-negotiates a new login, which provides an opportunity to potentially redirect the user's activity to a new Region.

Compute-layer request routing

With compute-layer request routing, illustrated in the following diagram, the code that runs in the compute layer determines whether to process the request locally or pass it to a copy of itself that's running in another Region. When you use the *write to one Region* mode, the compute layer might detect that it's not the active Region and allow local read operations while forwarding all write operations to another Region. This compute layer code must be aware of data topology and routing rules, and enforce them reliably, based on the latest settings that specify which Regions are active for which data. The outer software stack within the Region doesn't have to be aware of how read and write requests are routed by the micro service. In a robust design, the receiving Region validates whether it is the current primary for the write operation. If it isn't, it generates an error that indicates that the global state needs to be corrected. The receiving Region might also buffer the write operation for a while if the primary Region is in the process of changing. In all cases, the compute stack in a Region writes only to its local DynamoDB endpoint, but the compute stacks might communicate with one another.

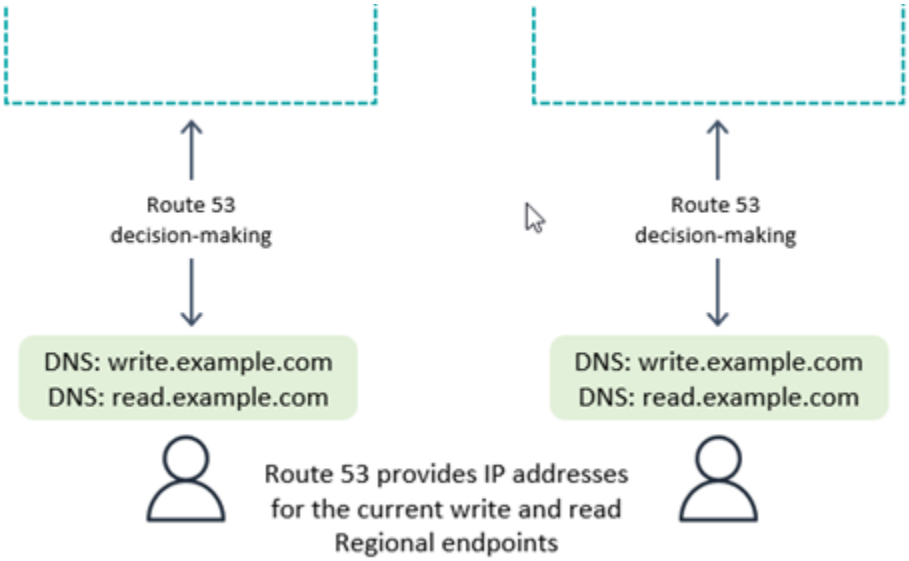


The Vanguard Group uses a system called Global Orchestration and Status Tool (GOaST) and a library called Global Multi-Region library (GMRlib) for this routing process, as presented at re:Invent 2022. They use a follow-the-sun single primary model. GOaST maintains the global state, similar to the ARC routing control discussed in the previous section. It uses a global table to track which Region is the primary Region and when the next primary switch is scheduled. All read and write operations go through GMRlib, which coordinates with GOaST. GMRlib allows read operations to be performed locally, at low latency. For write operations, GMRlib checks if the local Region is the current primary Region. If so, the write operation completes directly. If not, GMRlib forwards the write task to the GMRlib in the primary Region. That receiving library confirms that it also considers itself the primary Region and raises an error if it isn't, which indicates a propagation delay with the global state. This approach provides a validation benefit by not writing directly to a remote DynamoDB endpoint.

Route 53 request routing

Amazon Application Recovery Controller (ARC) is a Domain Name Service (DNS) technology. With Route 53, the client requests its endpoint by looking up a well-known DNS domain name,

and Route 53 returns the IP address corresponding to the regional endpoint(s) it thinks most appropriate. This is illustrated in the following diagram. Route 53 has a long list of routing policies it uses to determine the appropriate Region. It also can do failover routing to route traffic away from Regions that fail health checks.



With *write to any Region* mode, or if combined with the compute-layer request routing on the backend, Route 53 can be given full access to return the Region based on any complex internal rules such as the Region in closest network proximity, or closest geographic proximity, or any other choice.

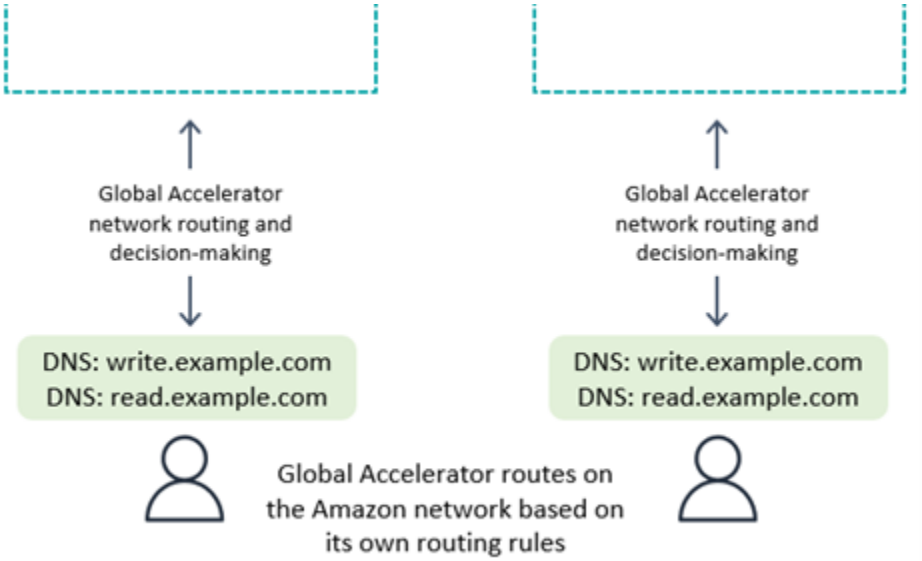
With *write to one Region* mode, you can configure Route 53 to return the currently active Region (using Route 53 ARC). If the client wants to connect to a passive Region (for example, for read operations), it could look up a different DNS name.

> **Note**
>
> Clients cache the IP addresses in the response from Route 53 for a time indicated by the time to live (TTL) setting on the domain name. A longer TTL extends the recovery time objective (RTO) for all clients to recognize the new endpoint. A value of 60 seconds is typical for failover use. Not all software perfectly adheres to DNS TTL expiration, and there might be multiple levels of DNS caching, such as at the operating system, virtual machine, and application.

With *write to your Region* mode, it's best to avoid Route 53 unless you're also using compute-layer request routing.

Global Accelerator request routing

With AWS Global Accelerator, illustrated in the following diagram, a client looks up the well-known domain name in Route 53. However, instead of getting back an IP address that corresponds to a Regional endpoint, the client receives an anycast static IP address which routes to the nearest AWS edge location. Starting from that edge location, all traffic gets routed on the private AWS network and to some endpoint (such as a load balancer or API Gateway) in a Region chosen by routing rules that are maintained within Global Accelerator. Compared with routing based on Route 53 rules, Global Accelerator request routing has lower latencies because it reduces the amount of traffic on the public internet. In addition, because Global Accelerator doesn't depend on DNS TTL expiration to change routing rules, it can adjust routing more quickly.



With *write to any Region* mode, or if combined with the compute-layer request routing on the back-end, Global Accelerator works seamlessly. The client connects to the nearest edge location and need not be concerned with which Region receives the request.

With *write to one Region* Global Accelerator routing rules must send requests to the currently active Region. You can use health checks that artificially report a failure on any Region that's not considered by your global system to be the active Region. As with DNS, it's possible to use an alternative DNS domain name for routing read requests if the requests can be from any Region.

With *write to your Region* mode, it's best to avoid Global Accelerator unless you're also using compute-layer request routing.

Evacuation processes

Evacuating a Region is the process of migrating activity, usually read and write activity or read activity, away from that Region.

Evacuating a live Region

You might decide to evacuate a live Region for a number of reasons: as part of usual business activity (for example, if you're using a follow-the-sun, write to one Region mode), due to a business decision to change the currently active Region, in response to failures in the software stack outside DynamoDB, or because you're encountering general issues such as higher than usual latencies within the Region.

With *write to any Region* mode, evacuating a live Region is straightforward. You can route traffic to alternative Regions by using any routing system and let the write operations in the evacuated Region replicate over as usual.

The write to one Region and write to your Region modes are usually used with MREC tables. Therefore, you must make sure that all write operations to the active Region have been fully recorded, stream processed, and globally propagated before starting write operations in the new active Region, to ensure that future write operations are processed against the latest version of the data.

Let's say that Region A is active and Region B is passive (either for the full table or for items that are homed in Region A). The typical mechanism to perform an evacuation is to pause write operations to A, wait long enough for those operations to have fully propagated to B, update the architecture stack to recognize B as active, and then resume write operations to B. There is no metric to indicate with absolute certainty that Region A has fully replicated its data to Region B. If Region A is healthy, pausing write operations to Region A and waiting 10 times the recent maximum value of the `ReplicationLatency` metric would typically be sufficient to determine that replication is complete. If Region A is unhealthy and shows other areas of increased latencies, you would choose a larger multiple for the wait time.

Evacuating an offline Region

There's a special case to consider: What if Region A goes fully offline without notice? This is extremely unlikely but should be considered nevertheless.

Evacuating an offline MRSC table

If this happens with an MRSC table, there is nothing special you need to do. MRSC tables support a recovery point objective (RPO) of zero. All successful write operations made to the MRSC table in the offline Region will be available in all other Region tables, so there's no potential gap in data even if the Region goes fully offline without notice. Business can continue using replicas located in the other Regions.

Evacuating an offline MREC table

If this happens with an MREC table, any write operations in Region A that were not yet propagated are held and propagated after Region A comes back online. The write operations aren't lost, but their propagation is indefinitely delayed.

How to proceed in this event is the application's decision. For business continuity, write operations might need to proceed to the new primary Region B. However, if an item in Region B receives an update while there is a pending propagation of a write operation for that item from Region A, the propagation is suppressed under the *last writer wins* model. Any update in Region B might suppress an incoming write request.

With the *write to any Region* mode, read and write operations can continue in Region B, trusting that the items in Region A will propagate to Region B eventually and recognizing the potential for missing items until Region A comes back online. When possible, such as with idempotent write operations, you should consider replaying recent write traffic (for example, by using an upstream event source) to fill in the gap of any potentially missing write operations and let the last writer wins conflict resolution suppress the eventual propagation of the incoming write operation.

With the other write modes, you have to consider the degree to which work can continue with a slightly out-of-date view of the world. Some small duration of write operations, as tracked by `ReplicationLatency`, will be missing until Region A comes back online. Can business move forward? In some use cases it can, but in others it might not without additional mitigation mechanisms.

For example, imagine that you have to maintain an available credit balance without interruption even after a full outage of a Region. You could split the balance into two different items, one homed in Region A and one in Region B, and start each with half the available balance. This would use the *write to your Region* mode. Transactional updates processed in each Region would write against the local copy of the balance. If Region A goes fully offline, work could still proceed with transaction processing in Region B, and write operations would be limited to the

balance portion held in Region B. Splitting the balance like this introduces complexities when the balance gets low or the credit has to be rebalanced, but it does provide one example of safe business recovery even with uncertain pending write operations.

As another example, imagine that you're capturing web form data. You can use Optimistic Concurrency Control (OCC) (OCC) to assign versions to data items and embed the latest version into the web form as a hidden field. On each submit, the write operation succeeds only if the version in the database still matches the version that the form was built against. If the versions don't match, the web form can be refreshed (or carefully merged) based on the current version in the database, and the user can proceed again. The OCC model usually protects against another client overwriting and producing a new version of the data, but it can also help during failover where a client might encounter older versions of data. Let's imagine that you're using the timestamp as the version. The form was first built against Region A at 12:00 but (after failover) tries to write to Region B and notices that the latest version in the database is 11:59. In this scenario, the client can either wait for the 12:00 version to propagate to Region B and then write on top of that version, or build on 11:59 and create a new 12:01 version (which, after writing, would suppress the incoming version after Region A recovers).

As a third example, a financial services company holds data about customer accounts and their financial transactions in a DynamoDB database. In the event of a complete Region A outage, they want to make sure that any write activity related to their accounts is either fully available in Region B, or they want to quarantine their accounts as known partial until Region A comes back online. Instead of pausing all business, they decided to pause business only to the tiny fraction of accounts that they determined had unpropagated transactions. To achieve this, they used a third Region, which we will call Region C. Before they processed any write operations in Region A, they placed a succinct summary of those pending operations (for example, a new transaction count for an account) in Region C. This summary was sufficient for Region B to determine if its view was fully up to date. This action effectively locked the account from the time of writing in Region C until Region A accepted the write operations and Region B received them. The data in Region C wasn't used except as part of a failover process, after which Region B could cross-check its data with Region C to check if any of its accounts were out of date. Those accounts would be marked as quarantined until the Region A recovery propagated the partial data to Region B. If Region C were to fail, a new Region D could be spun up for use instead. The data in Region C was very transient, and after a few minutes Region D would have a sufficiently up-to-date record of the in-flight write operations to be fully useful. If Region B were to fail, Region A could continue accepting write requests in cooperation with Region C. This company was willing to accept higher latency writes (to two Regions: C and then A)

and was fortunate to have a data model where the state of an account could be succinctly summarized.

Throughput capacity planning for DynamoDB global tables

Migrating traffic from one Region to another requires careful consideration of DynamoDB table settings regarding capacity.

Here are some considerations for managing write capacity:

- A global table must be in on-demand mode or provisioned with auto scaling enabled.
- If provisioned with auto scaling, the write settings (minimum, maximum, and target utilization) are replicated across Regions. Although the auto scaling settings are synchronized, the actual provisioned write capacity can float independently between Regions.
- One reason you could see different provisioned write capacity is due to the TTL feature. When you enable TTL in DynamoDB, you can specify an attribute name whose value indicates the time of expiration for the item, in Unix epoch time format in seconds. After that time, DynamoDB can delete the item without incurring write costs. With global tables, you can configure TTL in any Region, and the setting is automatically replicated to other Regions that are associated with the global table. When an item is eligible for deletion through a TTL rule, that work can be done in any Region. The delete operation is performed without consuming write units on the source table, but the replica tables will get a replicated write of that delete operation and will incur replicated write unit costs. TTL isn't supported in MRSC tables.
- If you’re using auto scaling, make sure that the maximum provisioned write capacity setting is sufficiently high to handle all write operations as well as all potential TTL delete operations. Auto scaling adjusts each Region according to its write consumption. On-demand tables have no maximum provisioned write capacity setting, but the *table-level maximum write throughput limit* specifies the maximum sustained write capacity the on-demand table will allow. The default limit to 40,000, but it is adjustable. We recommend that you set it high enough to handle all write operations (including TTL write operations) that the on-demand table might need. This value must be the same across all participating Regions when you set up global tables.

Here are some considerations for managing read capacity:

- Read capacity management settings are allowed to differ between Regions because it’s assumed that different Regions might have independent read patterns. When you first add a global replica

to a table, the capacity of the source Region is propagated. After creation you can adjust the read capacity settings, which aren't transferred to the other side.

- When you use DynamoDB auto scaling, make sure that the maximum provisioned read capacity settings are sufficiently high to handle all read operations across all Regions. During standard operations the read capacity will perhaps be spread across Regions, but during failover the table should be able to automatically adapt to the increased read workload. On-demand tables have no maximum provisioned read capacity setting, but the *table-level maximum read throughput limit* specifies the maximum sustained read capacity the on-demand table will allow. The default limit is 40,000, but it is adjustable. We recommend that you set it high enough to handle all read operations that the table might need if all read operations were to route to this single Region.
- If a table in one Region doesn't usually receive read traffic but might have to absorb a large amount of read traffic after a failover, you can pre-warm the capacity of the to accept a higher level of read traffic.

ARC has readiness checks that can be useful for confirming that DynamoDB Regions have similar table settings and account quotas, whether or not you use Route 53 to route requests. These readiness checks can also help in adjusting account-level quotas to make sure they match.

Preparation checklist for DynamoDB global tables

Use the following checklist for decisions and tasks when you deploy global tables.

- Determine if your use case benefits more from an MRSC or MREC consistency mode. Do you need strong consistency, even with the higher latency and other tradeoffs?
- Determine how many and which Regions should participate in the global table. If you plan to use MRSC, decide if you want the third Region to be a replica or a witness.
- Determine your application's write mode. This is not the same as the consistency mode. For more information, see Write modes with DynamoDB global tables.
- Plan your Routing strategies in DynamoDB strategy, based on your write mode.
- Define your

  Evacuating a Region is the process of migrating activity, usually read and write activity or read activity, away from that Region.

Evacuating a live Region

You might decide to evacuate a live Region for a number of reasons: as part of usual business activity (for example, if you're using a follow-the-sun, write to one Region mode), due to a business decision to change the currently active Region, in response to failures in the software stack outside DynamoDB, or because you're encountering general issues such as higher than usual latencies within the Region.

With *write to any Region* mode, evacuating a live Region is straightforward. You can route traffic to alternative Regions by using any routing system and let the write operations in the evacuated Region replicate over as usual.

The write to one Region and write to your Region modes are usually used with MREC tables. Therefore, you must make sure that all write operations to the active Region have been fully recorded, stream processed, and globally propagated before starting write operations in the new active Region, to ensure that future write operations are processed against the latest version of the data.

Let's say that Region A is active and Region B is passive (either for the full table or for items that are homed in Region A). The typical mechanism to perform an evacuation is to pause write operations to A, wait long enough for those operations to have fully propagated to B, update the architecture stack to recognize B as active, and then resume write operations to B. There is no metric to indicate with absolute certainty that Region A has fully replicated its data to Region B. If Region A is healthy, pausing write operations to Region A and waiting 10 times the recent maximum value of the `ReplicationLatency` metric would typically be sufficient to determine that replication is complete. If Region A is unhealthy and shows other areas of increased latencies, you would choose a larger multiple for the wait time.

Evacuating an offline Region

There's a special case to consider: What if Region A goes fully offline without notice? This is extremely unlikely but should be considered nevertheless.

Evacuating an offline MRSC table

If this happens with an MRSC table, there is nothing special you need to do. MRSC tables support a recovery point objective (RPO) of zero. All successful write operations made to the MRSC table in the offline Region will be available in all other Region tables, so there's

no potential gap in data even if the Region goes fully offline without notice. Business can continue using replicas located in the other Regions.

Evacuating an offline MREC table

If this happens with an MREC table, any write operations in Region A that were not yet propagated are held and propagated after Region A comes back online. The write operations aren't lost, but their propagation is indefinitely delayed.

How to proceed in this event is the application's decision. For business continuity, write operations might need to proceed to the new primary Region B. However, if an item in Region B receives an update while there is a pending propagation of a write operation for that item from Region A, the propagation is suppressed under the *last writer wins* model. Any update in Region B might suppress an incoming write request.

With the *write to any Region* mode, read and write operations can continue in Region B, trusting that the items in Region A will propagate to Region B eventually and recognizing the potential for missing items until Region A comes back online. When possible, such as with idempotent write operations, you should consider replaying recent write traffic (for example, by using an upstream event source) to fill in the gap of any potentially missing write operations and let the last writer wins conflict resolution suppress the eventual propagation of the incoming write operation.

With the other write modes, you have to consider the degree to which work can continue with a slightly out-of-date view of the world. Some small duration of write operations, as tracked by `ReplicationLatency`, will be missing until Region A comes back online. Can business move forward? In some use cases it can, but in others it might not without additional mitigation mechanisms.

For example, imagine that you have to maintain an available credit balance without interruption even after a full outage of a Region. You could split the balance into two different items, one homed in Region A and one in Region B, and start each with half the available balance. This would use the *write to your Region* mode. Transactional updates processed in each Region would write against the local copy of the balance. If Region A goes fully offline, work could still proceed with transaction processing in Region B, and write operations would be limited to the balance portion held in Region B. Splitting the balance like this introduces complexities when the balance gets low or the credit has to be rebalanced, but it does provide one example of safe business recovery even with uncertain pending write operations.

As another example, imagine that you're capturing web form data. You can use Optimistic Concurrency Control (OCC) (OCC) to assign versions to data items and embed the latest version into the web form as a hidden field. On each submit, the write operation succeeds only if the version in the database still matches the version that the form was built against. If the versions don't match, the web form can be refreshed (or carefully merged) based on the current version in the database, and the user can proceed again. The OCC model usually protects against another client overwriting and producing a new version of the data, but it can also help during failover where a client might encounter older versions of data. Let's imagine that you're using the timestamp as the version. The form was first built against Region A at 12:00 but (after failover) tries to write to Region B and notices that the latest version in the database is 11:59. In this scenario, the client can either wait for the 12:00 version to propagate to Region B and then write on top of that version, or build on 11:59 and create a new 12:01 version (which, after writing, would suppress the incoming version after Region A recovers).

As a third example, a financial services company holds data about customer accounts and their financial transactions in a DynamoDB database. In the event of a complete Region A outage, they want to make sure that any write activity related to their accounts is either fully available in Region B, or they want to quarantine their accounts as known partial until Region A comes back online. Instead of pausing all business, they decided to pause business only to the tiny fraction of accounts that they determined had unpropagated transactions. To achieve this, they used a third Region, which we will call Region C. Before they processed any write operations in Region A, they placed a succinct summary of those pending operations (for example, a new transaction count for an account) in Region C. This summary was sufficient for Region B to determine if its view was fully up to date. This action effectively locked the account from the time of writing in Region C until Region A accepted the write operations and Region B received them. The data in Region C wasn't used except as part of a failover process, after which Region B could cross-check its data with Region C to check if any of its accounts were out of date. Those accounts would be marked as quarantined until the Region A recovery propagated the partial data to Region B. If Region C were to fail, a new Region D could be spun up for use instead. The data in Region C was very transient, and after a few minutes Region D would have a sufficiently up-to-date record of the in-flight write operations to be fully useful. If Region B were to fail, Region A could continue accepting write requests in cooperation with Region C. This company was willing to accept higher latency writes (to two Regions: C and then A) and

was fortunate to have a data model where the state of an account could be succinctly summarized.

the section called “Evacuation processes”, based on your consistency mode, write mode, and routing strategy.

- Capture metrics on the health, latency, and errors across each Region. For a list of DynamoDB metrics, see the AWS blog post Monitoring Amazon DynamoDB for Operational Awareness for a list of metrics to observe. You should also use synthetic canaries (artificial requests designed to detect failures, named after the canary in the coal mine), as well as live observation of customer traffic. Not all issues will appear in the DynamoDB metrics.
- If you're using MREC, set alarms for any sustained increase in `ReplicationLatency`. An increase might indicate an accidental misconfiguration in which the global table has different write settings in different Regions, which leads to failed replicated requests and increased latencies. It could also indicate that there is a Regional disruption. A good example is to generate an alert if the recent average exceeds 180,000 milliseconds. You might also watch for `ReplicationLatency` dropping to 0, which indicates stalled replication.
- Assign sufficient maximum read and write settings for each global table.
- Identify the reasons for evacuating a Region in advance. If the decision involves human judgment, document all considerations. This work should be done carefully in advance, not under stress.
- Maintain a runbook for every action that must take place when you evacuate a Region. Usually very little work is involved for the global tables, but moving the rest of the stack might be complex.

  > **Note**
  >
  > With failover procedures, it's best practice to rely only on data plane operations and not on control plane operations, because some control plane operations could be degraded during Region failures.

  For more information, see the AWS blog post Build resilient applications with Amazon DynamoDB global tables: Part 4.
- Test all aspects of the runbook periodically, including Region evacuations. An untested runbook is an unreliable runbook.

- Consider using AWS Resilience Hub to evaluate the resilience of your entire application (including global tables). It provides a comprehensive view of your overall application portfolio resilience status through its dashboard.
- Consider using ARC readiness checks to evaluate the current configuration of your application and track any deviances from best practices.
- When you write health checks for use with Route 53 or Global Accelerator, make a set of calls that cover the full database flow. If you limit your check to confirm only that the DynamoDB endpoint is up, you won't be able to cover many failure modes such as AWS Identity and Access Management (IAM) configuration errors, code deployment problems, failure in the stack outside DynamoDB, higher than average read or write latencies, and so on.

Frequently Asked Questions (FAQ) for deploying global tables

What is the pricing for global tables?

- A write operation in a traditional DynamoDB table is priced in write capacity units (WCUs, for provisioned tables) or write request units (WRUs) for on-demand tables. If you write a 5 KB item, it incurs a charge of 5 units. A write to a global table is priced in replicated write capacity units (rWCUs, for provisioned tables) or replicated write request units (rWRUs, for on-demand tables). rWCUs and rWRUs are priced the same as WGUs and WRUs.
- rWCU and rWRU changes are incurred in every Region where the item is written directly or written through replication. Cross-Region data transfer fees apply.
- Writing to a global secondary index (GSI) is considered a local write operation and uses regular write units.
- There is no reserved capacity available for rWCUs or rWRUs at this time. Purchasing reserved capacity for WCUs can be beneficial for tables where GSIs consume write units.
- When you add a new Region to a global table, DynamoDB bootstraps the new Region automatically and charges you as if it were a table restore, based on the GB size of the table. It also charges cross-Region data transfer fees.

Which Regions does global tables support?

Global Tables version 2019.11.21 (Current) supports all AWS Regions for MREC tables and the following Region sets for MRSC tables:

- US Region set: US East (N.Virginia), US East (Ohio), US West (Oregon)

- EU Region set: Europe (Ireland), Europe (London), Europe (Paris), Europe (Frankfort)
- AP Region set: Asia Pacific (Tokyo), Asia Pacific (Seoul), and Asia Pacific (Osaka)

How are GSIs handled with global tables?

In Global Tables version 2019.11.21 (Current), when you create a GSI in one Region it's automatically created in other participating Regions and automatically backfilled.

How do I stop replication of a global table?

- You can delete a replica table the same way you would delete any other table. Deleting the global table stops replication to that Region and deletes the table copy kept in that Region. However, you can't stop replication while keeping copies of the table as independent entities, nor can you pause replication.
- An MRSC table must be deployed in exactly three Regions. To delete the replicas you must delete all the replicas and the witness so that the MRSC table becomes a local table.

How do DynamoDB Streams interact with global tables?

- Each global table produces an independent stream based on all its write operations, wherever they started from. You can choose to consume the DynamoDB stream in one Region or in all Regions (independently). If you want to process local but not replicated write operations, you can add your own Region attribute to each item to identify the writing Region. You can then use a Lambda event filter to call the Lambda function only for write operations in the local Region. This helps with insert and update operations, but not delete operations.
- Global tables that are configured for multi-Region eventual consistency (MREC tables) replicate changes by reading those changes from a DynamoDB stream on a replica table and applying that change to all other replica tables. Therefore, DynamoDB is enabled by default on all replicas in an MREC global table and cannot be disabled on those replicas. The MREC replication process can combine multiple changes in a short period of time into a single replicated write operation. As a result, each replica's stream might contain slightly different records. DynamoDB Streams records on MREC replicas are always ordered on a per-item basis, but ordering between items might differ between replicas.
- Global tables that are configured for multi-Region strong consistency (MRSC tables) don't use DynamoDB Streams for replication, so this feature isn't enabled by default on MRSC replicas. You can enable DynamoDB Streams on an MRSC replica. DynamoDB Streams records on MRSC

replicas are identical for every replica and are always ordered on a per-item basis, but ordering between items might differ between replicas.

How do global tables handle transactions?

- Transactional operations on MRSC tables will generate errors.
- Transactional operations on MREC tables provide atomicity, consistency, isolation, and durability (ACID) guarantees only within the Region where the write operation originally occurred. Transactions are not supported across Regions in global tables. For example, if you have an MREC global table with replicas in the US East (Ohio) and US West (Oregon) Regions and perform a `TransactWriteItems` operation in the US East (Ohio) Region, you might observe partially completed transactions in the US West (Oregon) Region as changes are replicated. Changes are replicated to other Regions only after they have been committed in the source Region.

How do global tables interact with the DynamoDB Accelerator cache (DAX)?

Global tables bypass DAX by updating DynamoDB directly, so DAX isn't aware that it's holding stale data. The DAX cache is refreshed only when the cache's TTL expires.

Do tags on tables propagate?

No, tags do not automatically propagate.

Should I backup tables in all Regions or just one?

The answer depends on the purpose of the backup.

- If you want to ensure data durability, DynamoDB already provides that safeguard. The service ensures durability.
- If you want to keep a snapshot for historical records (for example, to meet regulatory requirements), backing up in one Region should suffice. You can copy the backup to additional Regions by using AWS Backup.
- If you want to recover erroneously deleted or modified data, use DynamoDB point-in-time recovery (PITR) in one Region.

How do I deploy global tables using CloudFormation?

- CloudFormation represents a DynamoDB table and a global table as two separate resources: `AWS::DynamoDB::Table` and `AWS::DynamoDB::GlobalTable`. One approach is to create all tables that can potentially be global by using the `GlobalTable` construct of keeping them as standalone tables initially, and add Regions later if necessary.
- In CloudFormation, each global table is controlled by a single stack, in a single Region, regardless of the number of replicas. When you deploy your template, CloudFormation creates and updates all replicas as part of a single stack operation. You should not deploy the same AWS::DynamoDB::GlobalTable resource in multiple Regions. This will result in errors and is unsupported. If you deploy your application template in multiple Regions, you can use conditions to create the `AWS::DynamoDB::GlobalTable` resource in a single Region. Alternatively, you can choose to define your `AWS::DynamoDB::GlobalTable` resources in a stack that's separate from your application stack, and make sure that it's deployed to a single Region.
- If you have a regular table and you want to convert it to a global table while keeping it managed by CloudFormation then set the deletion policy to `Retain`, remove the table from the stack, convert the table to a global table in the console, and then import the global table as a new resource to the stack. For more information, see the AWS GitHub repository.
- Cross-account replication is not supported at this time.

Conclusion and resources

DynamoDB global tables have very few controls but still require careful consideration. You must determine your write mode, routing model, and evacuation processes. You must instrument your application across every Region and be ready to adjust your routing or perform an evacuation to maintain global health. The reward is having a globally distributed dataset with low-latency read and write operations that is designed for 99.999% availability.

For more information about DynamoDB global tables, see the following resources:

- DynamoDB documentation
- Amazon Application Recovery Controller
- Readiness check in ARC (AWS documentation)
- Route 53 routing policies
- AWS Global Accelerator
- DynamoDB service-level agreement
- AWS Multi-Region Fundamentals (AWS whitepaper)

- Data resiliency design patterns with AWS (AWS re:Invent 2022 presentation)
- How Fidelity Investments and Reltio modernized with Amazon DynamoDB (AWS re:Invent 2022 presentation)
- Multi-Region design patterns and best practices (AWS re:Invent 2022 presentation)
- Disaster Recovery (DR) Architecture on AWS, Part III: Pilot Light and Warm Standby (AWS blog post)
- Use Region pinning to set a home Region for items in an Amazon DynamoDB global table (AWS blog post)
- Monitoring Amazon DynamoDB for operational awareness (AWS blog post)
- Scaling DynamoDB: How partitions, hot keys, and split for heat impact performance (AWS blog post)
- Multi-Region strong consistency with DynamoDB global tables (AWS re:Invent 2024 presentation)

Best practices for managing the control plane in DynamoDB

> **Note**
>
> DynamoDB is introducing a control plane throttle limit of 2,500 requests per second with the option for a retry. See below for additional details.

DynamoDB control plane operations let you manage DynamoDB tables as well as objects that are dependent on tables such as indexes. For more information about these operations, see Control plane.

In some circumstances, you may need to take actions and use data returned by control plane calls as part of your business logic. For example, you might need to know the value of `ProvisionedThroughput` returned by `DescribeTable`. In these circumstances, follow these best practices:

- Do not excessively query the DynamoDB control plane.
- Do not mix control plane calls and data plane calls within the same code.
- Handle throttles on control plane requests and retry with a backoff.
- Invoke and track changes to a particular resource from a single client.

- Instead of retrieving data for the same table multiple times at short intervals, cache the data for processing.

Best practices for using bulk data operations in DynamoDB

DynamoDB supports batch operations such as `BatchWriteItem` using which you can perform up to 25 `PutItem` and `DeleteItem` requests together. However, `BatchWriteItem` doesn't support `UpdateItem` operations. When it comes to bulk updates, the distinction lies in the requirements and the nature of the update. You can use other DynamoDB APIs such as `TransactWriteItems` for batch size up to 100. When more items are involved, you can use services such as AWS Glue, Amazon EMR, AWS Step Functions or use custom scripts and tools like DynamoDB-shell for bulk updates.

Topics

- Conditional batch update
- Efficient bulk operations

Conditional batch update

DynamoDB supports batch operations such as `BatchWriteItem` using which you can perform up to 25 `PutItem` and `DeleteItem` requests in a single batch. However, `BatchWriteItem` doesn't support `UpdateItem` operations and doesn't support condition expressions. As a workaround, you can use other DynamoDB APIs such as `TransactWriteItems` for batch size up to 100.

When more items are involved, and a major chunk of data needs to be changed, you can use services such as AWS Glue, Amazon EMR, AWS Step Functions or use custom scripts and tools like DynamoDB-shell for efficient bulk updates.

When to use this pattern

- DynamoDB-shell is not a supported for production use case.
- `TransactWriteItems` – up to 100 individual updates with or without conditions, executing as an all or nothing ACID bundle. `TransactWriteItems` calls can also be supplied with a `ClientRequestToken` if your application requires idempotency, meaning multiple identical calls have the same effect as one single call. This ensures you don't execute the same transaction multiple times and end up with an incorrect state of data.

Trade-off – Additional throughput is consumed. 2 WCUs per 1KB write instead of the standard 1 WGU per 1 KB write.

- PartiQL `BatchExecuteStatement` – up to 25 updates with or without conditions. `BatchExecuteStatement` always returns a success response to the overall request, and also returns a list of individual operation responses that preserves order.

  Trade-off – For larger batches, additional client-side logic is required to distribute requests in batches of 25. Individual error responses need to be considered to determine retry strategy.

Code examples

These code examples use the boto3 library, which is the AWS SDK for Python. The examples assume you have boto3 installed and configured with appropriate AWS credentials.

Assume an inventory database for an electrical appliance vendor who has multiple warehouses across European cities. Because it is end of summer, the vendor would like to clear out desk fans to make room for other stock. The vendor wants to provide a price discount for all desk fans supplied out of warehouses in Italy but only if they have a reserve stock of 20 desk fans. The DynamoDB table is called **inventory**, it has a key schema of Partition key **sku** which is a unique identifier for each product and a Sort key **warehouse** which is an identifier for a warehouse.

The following Python code demonstrates how to perform this conditional batch update using `BatchExecuteStatement` API call.

```
import boto3

client=boto3.client("dynamodb")

before_image=client.query(TableName='inventory', KeyConditionExpression='sku=:pk_val
 AND begins_with(warehouse, :sk_val)', ExpressionAttributeValues={':pk_val':
{'S':'F123'},':sk_val':{'S':'WIT'}},
 ProjectionExpression='sku,warehouse,quantity,price')
print("Before update: ", before_image['Items'])

response=client.batch_execute_statement(
        Statements=[
            {'Statement': 'UPDATE inventory SET price=price-5 WHERE sku=? AND
 warehouse=? AND quantity > 20', 'Parameters': [{'S':'F123'}, {'S':'WITTUR1'}],
 'ReturnValuesOnConditionCheckFailure': 'ALL_OLD'},
```

```
            {'Statement': 'UPDATE inventory SET price=price-5 WHERE sku=? AND
 warehouse=? AND quantity > 20', 'Parameters': [{'S':'F123'}, {'S':'WITROM1'}],
 'ReturnValuesOnConditionCheckFailure': 'ALL_OLD'},
            {'Statement': 'UPDATE inventory SET price=price-5 WHERE sku=? AND
 warehouse=? AND quantity > 20', 'Parameters': [{'S':'F123'}, {'S':'WITROM2'}],
 'ReturnValuesOnConditionCheckFailure': 'ALL_OLD'},
            {'Statement': 'UPDATE inventory SET price=price-5 WHERE sku=? AND
 warehouse=? AND quantity > 20', 'Parameters': [{'S':'F123'}, {'S':'WITROM5'}],
 'ReturnValuesOnConditionCheckFailure': 'ALL_OLD'},
            {'Statement': 'UPDATE inventory SET price=price-5 WHERE sku=? AND
 warehouse=? AND quantity > 20', 'Parameters': [{'S':'F123'}, {'S':'WITVEN1'}],
 'ReturnValuesOnConditionCheckFailure': 'ALL_OLD'},
            {'Statement': 'UPDATE inventory SET price=price-5 WHERE sku=? AND
 warehouse=? AND quantity > 20', 'Parameters': [{'S':'F123'}, {'S':'WITVEN2'}],
 'ReturnValuesOnConditionCheckFailure': 'ALL_OLD'},
            {'Statement': 'UPDATE inventory SET price=price-5 WHERE sku=? AND
 warehouse=? AND quantity > 20', 'Parameters': [{'S':'F123'}, {'S':'WITVEN3'}],
 'ReturnValuesOnConditionCheckFailure': 'ALL_OLD'},
        ],
        ReturnConsumedCapacity='TOTAL'
    )

after_image=client.query(TableName='inventory', KeyConditionExpression='sku=:pk_val
 AND begins_with(warehouse, :sk_val)', ExpressionAttributeValues={':pk_val':
{'S':'F123'},':sk_val':{'S':'WIT'}},
 ProjectionExpression='sku,warehouse,quantity,price')
print("After update: ", after_image['Items'])
```

Execution produces the below output on sample data:

```
Before update:  [{'quantity': {'N': '20'}, 'warehouse': {'S': 'WITROM1'}, 'price':
 {'N': '40'}, 'sku': {'S': 'F123'}}, {'quantity': {'N': '25'}, 'warehouse': {'S':
 'WITROM2'}, 'price': {'N': '40'}, 'sku': {'S': 'F123'}}, {'quantity': {'N':
 '28'}, 'warehouse': {'S': 'WITROM5'}, 'price': {'N': '38'}, 'sku': {'S': 'F123'}},
 {'quantity': {'N': '26'}, 'warehouse': {'S': 'WITTUR1'}, 'price': {'N': '40'}, 'sku':
 {'S': 'F123'}}, {'quantity': {'N': '10'}, 'warehouse': {'S': 'WITVEN1'}, 'price':
 {'N': '38'}, 'sku': {'S': 'F123'}}, {'quantity': {'N': '20'}, 'warehouse': {'S':
 'WITVEN2'}, 'price': {'N': '38'}, 'sku': {'S': 'F123'}}, {'quantity': {'N': '50'},
 'warehouse': {'S': 'WITVEN3'}, 'price': {'N': '35'}, 'sku': {'S': 'F123'}}]
After update:  [{'quantity': {'N': '20'}, 'warehouse': {'S': 'WITROM1'}, 'price': {'N':
 '40'}, 'sku': {'S': 'F123'}}, {'quantity': {'N': '25'}, 'warehouse': {'S': 'WITROM2'},
 'price': {'N': '35'}, 'sku': {'S': 'F123'}}, {'quantity': {'N': '28'}, 'warehouse':
```

```
 {'S': 'WITROM5'}, 'price': {'N': '33'}, 'sku': {'S': 'F123'}}, {'quantity': {'N':
 '26'}, 'warehouse': {'S': 'WITTUR1'}, 'price': {'N': '35'}, 'sku': {'S': 'F123'}},
 {'quantity': {'N': '10'}, 'warehouse': {'S': 'WITVEN1'}, 'price': {'N': '38'}, 'sku':
 {'S': 'F123'}}, {'quantity': {'N': '20'}, 'warehouse': {'S': 'WITVEN2'}, 'price':
 {'N': '38'}, 'sku': {'S': 'F123'}}, {'quantity': {'N': '50'}, 'warehouse': {'S':
 'WITVEN3'}, 'price': {'N': '30'}, 'sku': {'S': 'F123'}}]
```

Since this is a bounded operation for an internal system, idempotency requirements haven't been considered. It's possible to place additional guardrails like price update should go through only if price is greater than 35 and less than 40 to make the updates more robust.

Alternatively, we can perform the same batch update operation using `TransactWriteItems` in case of stricter idempotency and ACID requirements. However, it is important to remember that either all the operations in the transaction bundle go through or the entire bundle fails.

Let's assume a case where there is a heatwave in Italy and the demand for desk fans has increased sharply. The vendor wants to increase their desk fan cost going out of every warehouse in Italy by 20 Euros but the regulatory body only allows this cost increase if the current cost is less than 70 Euros across their entire inventory. It's essential that the price is updated throughout the inventory at once and only once and only if the cost is less than 70 Euros in each of their warehouse.

The following Python code demonstrates how to perform this batch update using `TransactWriteItems` API call.

```
import boto3

client=boto3.client("dynamodb")

before_image=client.query(TableName='inventory', KeyConditionExpression='sku=:pk_val
 AND begins_with(warehouse, :sk_val)', ExpressionAttributeValues={':pk_val':
{'S':'F123'},':sk_val':{'S':'WIT'}},
 ProjectionExpression='sku,warehouse,quantity,price')
print("Before update: ", before_image['Items'])

response=client.transact_write_items(
        ClientRequestToken='UUIDAWS124',
        TransactItems=[
            {'Update': { 'Key': {'sku': {'S':'F123'}, 'warehouse': {'S':'WITTUR1'}},
 'UpdateExpression': 'SET price = price + :inc', 'ConditionExpression': 'price < :cap',
```

```
 'ExpressionAttributeValues': { ':inc': {'N': '20'}, ':cap': {'N': '70'}}, 'TableName':
 'inventory', 'ReturnValuesOnConditionCheckFailure': 'ALL_OLD'}},
           {'Update': { 'Key': {'sku': {'S':'F123'}, 'warehouse': {'S':'WITROM1'}},
 'UpdateExpression': 'SET price = price + :inc', 'ConditionExpression': 'price < :cap',
 'ExpressionAttributeValues': { ':inc': {'N': '20'}, ':cap': {'N': '70'}}, 'TableName':
 'inventory', 'ReturnValuesOnConditionCheckFailure': 'ALL_OLD'}},
           {'Update': { 'Key': {'sku': {'S':'F123'}, 'warehouse': {'S':'WITROM2'}},
 'UpdateExpression': 'SET price = price + :inc', 'ConditionExpression': 'price < :cap',
 'ExpressionAttributeValues': { ':inc': {'N': '20'}, ':cap': {'N': '70'}}, 'TableName':
 'inventory', 'ReturnValuesOnConditionCheckFailure': 'ALL_OLD'}},
           {'Update': { 'Key': {'sku': {'S':'F123'}, 'warehouse': {'S':'WITROM5'}},
 'UpdateExpression': 'SET price = price + :inc', 'ConditionExpression': 'price < :cap',
 'ExpressionAttributeValues': { ':inc': {'N': '20'}, ':cap': {'N': '70'}}, 'TableName':
 'inventory', 'ReturnValuesOnConditionCheckFailure': 'ALL_OLD'}},
           {'Update': { 'Key': {'sku': {'S':'F123'}, 'warehouse': {'S':'WITVEN1'}},
 'UpdateExpression': 'SET price = price + :inc', 'ConditionExpression': 'price < :cap',
 'ExpressionAttributeValues': { ':inc': {'N': '20'}, ':cap': {'N': '70'}}, 'TableName':
 'inventory', 'ReturnValuesOnConditionCheckFailure': 'ALL_OLD'}},
           {'Update': { 'Key': {'sku': {'S':'F123'}, 'warehouse': {'S':'WITVEN2'}},
 'UpdateExpression': 'SET price = price + :inc', 'ConditionExpression': 'price < :cap',
 'ExpressionAttributeValues': { ':inc': {'N': '20'}, ':cap': {'N': '70'}}, 'TableName':
 'inventory', 'ReturnValuesOnConditionCheckFailure': 'ALL_OLD'}},
           {'Update': { 'Key': {'sku': {'S':'F123'}, 'warehouse': {'S':'WITVEN3'}},
 'UpdateExpression': 'SET price = price + :inc', 'ConditionExpression': 'price < :cap',
 'ExpressionAttributeValues': { ':inc': {'N': '20'}, ':cap': {'N': '70'}}, 'TableName':
 'inventory', 'ReturnValuesOnConditionCheckFailure': 'ALL_OLD'}},
       ],
       ReturnConsumedCapacity='TOTAL'
    )

after_image=client.query(TableName='inventory', KeyConditionExpression='sku=:pk_val
 AND begins_with(warehouse, :sk_val)', ExpressionAttributeValues={':pk_val':
{'S':'F123'},':sk_val':{'S':'WIT'}},
 ProjectionExpression='sku,warehouse,quantity,price')
print("After update: ", after_image['Items'])
```

Execution produces the below output on sample data:

```
Before update:  [{'quantity': {'N': '20'}, 'warehouse': {'S': 'WITROM1'}, 'price':
 {'N': '60'}, 'sku': {'S': 'F123'}}, {'quantity': {'N': '25'}, 'warehouse': {'S':
 'WITROM2'}, 'price': {'N': '55'}, 'sku': {'S': 'F123'}}, {'quantity': {'N':
 '28'}, 'warehouse': {'S': 'WITROM5'}, 'price': {'N': '53'}, 'sku': {'S': 'F123'}},
```

```
 {'quantity': {'N': '26'}, 'warehouse': {'S': 'WITTUR1'}, 'price': {'N': '55'}, 'sku':
 {'S': 'F123'}}, {'quantity': {'N': '10'}, 'warehouse': {'S': 'WITVEN1'}, 'price':
 {'N': '58'}, 'sku': {'S': 'F123'}}, {'quantity': {'N': '20'}, 'warehouse': {'S':
 'WITVEN2'}, 'price': {'N': '58'}, 'sku': {'S': 'F123'}}, {'quantity': {'N': '50'},
 'warehouse': {'S': 'WITVEN3'}, 'price': {'N': '50'}, 'sku': {'S': 'F123'}}]
After update:  [{'quantity': {'N': '20'}, 'warehouse': {'S': 'WITROM1'}, 'price': {'N':
 '80'}, 'sku': {'S': 'F123'}}, {'quantity': {'N': '25'}, 'warehouse': {'S': 'WITROM2'},
 'price': {'N': '75'}, 'sku': {'S': 'F123'}}, {'quantity': {'N': '28'}, 'warehouse':
 {'S': 'WITROM5'}, 'price': {'N': '73'}, 'sku': {'S': 'F123'}}, {'quantity': {'N':
 '26'}, 'warehouse': {'S': 'WITTUR1'}, 'price': {'N': '75'}, 'sku': {'S': 'F123'}},
 {'quantity': {'N': '10'}, 'warehouse': {'S': 'WITVEN1'}, 'price': {'N': '78'}, 'sku':
 {'S': 'F123'}}, {'quantity': {'N': '20'}, 'warehouse': {'S': 'WITVEN2'}, 'price':
 {'N': '78'}, 'sku': {'S': 'F123'}}, {'quantity': {'N': '50'}, 'warehouse': {'S':
 'WITVEN3'}, 'price': {'N': '70'}, 'sku': {'S': 'F123'}}]
```

There are multiple approaches to perform batch updates in DynamoDB. The suitable approach depends on factors such as ACID and/or idempotency requirements, number of items to be updated, and familiarity with APIs.

Efficient bulk operations

When to use this pattern

These patterns are useful to efficiently perform bulk updates on DynamoDB items.

- DynamoDB-shell is not a supported for production use case.
- `TransactWriteItems` – up to 100 individual updates with or without conditions, executing as an all or nothing ACID bundle

  Trade-off – Additional throughput is consumed, 2 WCUs per 1 KB write.
- PartiQL `BatchExecuteStatement` – up to 25 updates with or without conditions

  Trade-off – Additional logic is required to distribute requests in batches of 25.
- AWS Step Functions – rate-limited bulk operations for developers familiar with AWS Lambda.

  Trade-off – Execution time is inversely proportional to rate-limit. Limited by the maximum Lambda function timeout. The functionality entails that data changes that occur between the read and the write may be overwritten. For more info, see Backfilling an Amazon DynamoDB Time to Live attribute using Amazon EMR: Part 2.

- AWS Glue and Amazon EMR – rate-limited bulk operation with managed parallelism. For applications or updates that are not time-sensitive, these options can run in the background only consuming a small percentage of throughput. Both services uses the emr-dynamodb-connector to perform DynamoDB operations. These services perform a big read followed by a big write of updated items with an option to rate-limit.

  Trade-off – Execution time is inversely proportional to rate-limit. The functionality includes that data changes occurring between the read and the write can be overwritten. You can't read from Global Secondary Indexes (GSIs). See, Backfilling an Amazon DynamoDB Time to Live attribute using Amazon EMR: Part 2.
- DynamoDB Shell – rate-limited bulk operations using SQL-like queries. You can read from GSIs for better efficiency.

  Trade-off – Execution time is inversely proportional to rate-limit. See Rate limited bulk operations in DynamoDB Shell.

Using the pattern

Bulk updates can have significant cost implications especially if you use the on-demand throughput mode. There's a trade-off between speed and cost if you use the provisioned throughput mode. Setting the rate-limit parameter very strictly can lead to a very large processing time. You can roughly determine speed of the update using the average item size and the rate limit.

Alternatively, you can determine amount of throughput needed for the process based on the expected duration of the update process and the average item size. The blog references shared with each pattern provide details on the strategy, implementation and limitations of using the pattern. For more information, see Cost-effective bulk processing with Amazon DynamoDB.

There are multiple approaches to perform bulk-updates against a live DynamoDB table. The suitable approach depends on factors such as ACID and/or idempotency requirements, number of items to be updated and familiarity with APIs. It is important to consider the cost versus time trade-off, most approaches discussed above provide an option to rate-limit the throughput used by the bulk update job.

Best practices for handling concurrent updates in DynamoDB

In distributed systems, multiple processes or users may attempt to modify the same data at the same time. Without concurrency control, these concurrent writes can lead to lost updates,

inconsistent data, or race conditions. DynamoDB provides several mechanisms to help you manage concurrent access and maintain data integrity.

> **Note**
>
> Individual write operations such as `UpdateItem` are atomic and always operate on the most recent version of the item, regardless of concurrency. Locking strategies are needed when your application must read an item and then write it back based on the read value (a read-modify-write cycle), because another process could modify the item between the read and the write.

There are two primary strategies for handling concurrent updates:

- **Optimistic locking** – Assumes conflicts are rare. It allows concurrent access and detects conflicts at write time using conditional writes. If a conflict is detected, the write fails and the application can retry.
- **Pessimistic locking** – Assumes conflicts are likely. It prevents concurrent access by acquiring exclusive access to a resource before modifying it. Other processes must wait until the lock is released.

The following table summarizes the approaches available in DynamoDB:

Approach	Mechanism	Best for
Optimistic locking	Version attribute + conditional writes	Low contention, inexpensive retries
Pessimistic locking (transactions)	`TransactWriteItems`	Multi-item atomicity, moderate contention
Pessimistic locking (lock client)	Dedicated lock table with lease and heartbeat	Long-running workflows, distributed coordination

Optimistic locking with version number

Optimistic locking is a strategy that detects conflicts at write time rather than preventing them. Each item includes a version attribute that increments with every update. When updating an item, you include a condition expression that checks whether the version number matches the value your application last read. If another process modified the item in the meantime, the condition fails and DynamoDB returns a `ConditionalCheckFailedException`.

When to use optimistic locking

Optimistic locking is a good fit when:

- Multiple users or processes may update the same item, but conflicts are infrequent.
- Retrying a failed write is inexpensive for your application.
- You want to avoid the overhead and complexity of managing distributed locks.

Common examples include e-commerce inventory updates, collaborative editing platforms, and financial transaction records.

Tradeoffs

Retry overhead in high contention

In high-concurrency environments, the likelihood of conflicts increases, potentially causing higher retries and write costs.

Implementation complexity

Adding version control to items and handling conditional checks adds complexity to the application logic. The AWS SDK for Java v2 Enhanced Client provides built-in support through the `@DynamoDbVersionAttribute` annotation, which automatically manages version numbers for you.

Pattern design

Include a version attribute in each item. Here is a simple schema design:

- Partition key – A unique identifier for each item (for example, `ItemId`).
- Attributes:

- `ItemId` – The unique identifier for the item.
- `Version` – An integer that represents the version number of the item.
- `QuantityLeft` – The remaining inventory of the item.

When an item is first created, the `Version` attribute is set to 1. With each update, the version number increments by 1.

ItemID (partition key)	Version	QuantityLeft
Bananas	1	10
Apples	1	5
Oranges	1	7

Implementation

To implement optimistic locking, follow these steps:

1. Read the current version of the item.

```
def get_item(item_id):
    response = table.get_item(Key={'ItemID': item_id})
    return response['Item']

item = get_item('Bananas')
current_version = item['Version']
```

2. Update the item using a condition expression that checks the version number.

```
def update_item(item_id, qty_bought, current_version):
    try:
        response = table.update_item(
            Key={'ItemID': item_id},
            UpdateExpression="SET QuantityLeft = QuantityLeft - :qty, Version
 = :new_v",
            ConditionExpression="Version = :expected_v",
            ExpressionAttributeValues={
                ':qty': qty_bought,
```

```
                ':new_v': current_version + 1,
                ':expected_v': current_version
            },
            ReturnValues="UPDATED_NEW"
        )
        return response
    except ClientError as e:
        if e.response['Error']['Code'] == 'ConditionalCheckFailedException':
            print("Version conflict: another process updated this item.")
        raise
```

3. Handle conflicts by retrying with a fresh read.

   Each retry requires an additional read, so limit the total number of retries.

```
def update_with_retry(item_id, qty_bought, max_retries=3):
    for attempt in range(max_retries):
        item = get_item(item_id)
        try:
            return update_item(item_id, qty_bought, item['Version'])
        except ClientError as e:
            if e.response['Error']['Code'] != 'ConditionalCheckFailedException':
                raise
            print(f"Retry {attempt + 1}/{max_retries}")
    raise Exception("Update failed after maximum retries.")
```

For Java applications, the AWS SDK for Java v2 Enhanced Client provides built-in optimistic locking support through the `@DynamoDbVersionAttribute` annotation, which automatically manages version numbers for you.

For more information about condition expressions, see DynamoDB condition expression CLI example.

Pessimistic locking with DynamoDB transactions

DynamoDB transactions provide an all-or-nothing approach to grouped operations. When you use `TransactWriteItems`, DynamoDB monitors all items in the transaction. If any item is modified by another operation during the transaction, the entire transaction is canceled and DynamoDB returns a `TransactionCanceledException`. This behavior provides a form of pessimistic concurrency control because conflicting concurrent modifications are prevented rather than detected after the fact.

When to use transactions for locking

Transactions are a good fit when:

- You need to update multiple items atomically, either within the same table or across tables.
- Your business logic requires all-or-nothing semantics – either all changes succeed or none are applied.

Common examples include transferring funds between accounts, placing orders that update both inventory and order tables, and exchanging items between players in a game.

Tradeoffs

Higher write cost

For items up to 1 KB, transactions consume 2 WCUs per item (one to prepare, one to commit), compared to 1 WCU for a standard write.

Item limit

A single transaction can include up to 100 actions across one or more tables.

Conflict sensitivity

If any item in the transaction is modified by another operation, the entire transaction fails. In high-contention scenarios, this can lead to frequent cancellations.

Implementation

The following example uses `TransactWriteItems` to transfer inventory between two items atomically. If another process modifies either item during the transaction, the entire operation is rolled back.

```
import boto3

client = boto3.client('dynamodb')

def transfer_inventory(source_id, target_id, quantity):
    try:
        client.transact_write_items(
            TransactItems=[
                {
```

```
                    'Update': {
                        'TableName': 'Inventory',
                        'Key': {'ItemID': {'S': source_id}},
                        'UpdateExpression': 'SET QuantityLeft = QuantityLeft - :qty',
                        'ConditionExpression': 'QuantityLeft >= :qty',
                        'ExpressionAttributeValues': {
                            ':qty': {'N': str(quantity)}
                        }
                    }
                },
                {
                    'Update': {
                        'TableName': 'Inventory',
                        'Key': {'ItemID': {'S': target_id}},
                        'UpdateExpression': 'SET QuantityLeft = QuantityLeft + :qty',
                        'ExpressionAttributeValues': {
                            ':qty': {'N': str(quantity)}
                        }
                    }
                }
            ]
        )
        return True
    except client.exceptions.TransactionCanceledException as e:
        print(f"Transaction canceled: {e}")
        return False
```

In this example, the condition expression checks that sufficient inventory exists, but no version attribute is needed. DynamoDB automatically cancels the transaction if any item in the transaction is modified by another operation between the prepare and commit phases. This is what provides the pessimistic concurrency control – conflicting concurrent modifications are prevented by the transaction itself.

> **Note**
>
> You can combine transactions with optimistic locking by adding version checks as additional condition expressions. This provides an extra layer of protection but is not required for the transaction to detect conflicts.

For more information, see Managing complex workflows with DynamoDB transactions.

Distributed locking with the DynamoDB Lock Client

For applications that require traditional lock-acquire-release semantics, the DynamoDB Lock Client is an open-source library that implements distributed locking using a DynamoDB table as the lock store. This approach is useful when you need to coordinate access to an external resource (such as an S3 object or a shared configuration) across multiple application instances.

The lock client is available as an open-source Java library.

How it works

The lock client uses a dedicated DynamoDB table to track locks. Each lock is represented as an item with the following key attributes:

- A partition key that identifies the resource being locked.
- A lease duration that specifies how long the lock is valid. If the lock holder crashes or becomes unresponsive, the lock automatically expires after the lease duration.
- A heartbeat that the lock holder sends periodically to extend the lease. This prevents the lock from expiring while the holder is still actively processing.

The lock client uses conditional writes to ensure that only one process can acquire a lock at a time. If a lock is already held, the caller can choose to wait and retry or fail immediately.

When to use the lock client

The lock client is a good fit when:

- You need to coordinate access to a shared resource across multiple application instances or microservices.
- The critical section is long-running (seconds to minutes) and retrying the entire operation on conflict would be expensive.
- You need automatic lock expiry to handle process failures gracefully.

Common examples include orchestrating distributed workflows, coordinating cron jobs across multiple instances, and managing access to shared external resources.

Tradeoffs

Additional infrastructure

Requires a dedicated DynamoDB table for lock management, with additional read and write capacity for lock operations and heartbeats.

Clock dependency

Lock expiry relies on timestamps. Significant clock skew between clients can cause unexpected behavior, particularly for short lease durations.

Deadlock risk

If your application acquires locks on multiple resources, you must acquire them in a consistent order to avoid deadlocks. The lease duration provides a safety net by automatically releasing locks from unresponsive holders.

Implementation

The following example shows how to use the DynamoDB Lock Client to acquire and release a lock:

```
import java.io.IOException;
import java.util.Optional;
import java.util.concurrent.TimeUnit;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;

final DynamoDbClient dynamoDB = DynamoDbClient.builder()
    .region(Region.US_WEST_2)
    .build();

final AmazonDynamoDBLockClient lockClient = new AmazonDynamoDBLockClient(
    AmazonDynamoDBLockClientOptions.builder(dynamoDB, "Locks")
        .withTimeUnit(TimeUnit.SECONDS)
        .withLeaseDuration(10L)
        .withHeartbeatPeriod(3L)
        .withCreateHeartbeatBackgroundThread(true)
        .build());

// Try to acquire a lock on a resource
final Optional<LockItem> lock =
    lockClient.tryAcquireLock(AcquireLockOptions.builder("my-shared-
resource").build());
```

```
if (lock.isPresent()) {
    try {
        // Perform operations that require exclusive access
        processSharedResource();
    } finally {
        // Always release the lock when done
        lockClient.releaseLock(lock.get());
    }
} else {
    System.out.println("Failed to acquire lock.");
}

lockClient.close();
```

Important

Always release locks in a `finally` block to ensure locks are released even if your processing logic throws an exception. Unreleased locks block other processes until the lease expires.

You can also implement a simple locking mechanism without the lock client library by using conditional writes directly. The following example uses `UpdateItem` with a condition expression to acquire a lock, and `DeleteItem` to release it:

```
from datetime import datetime, timedelta
from boto3.dynamodb.conditions import Attr

def acquire_lock(table, resource_name, owner_id, ttl_seconds):
    """Attempt to acquire a lock. Returns True if successful."""
    expiry = (datetime.now() + timedelta(seconds=ttl_seconds)).isoformat()
    now = datetime.now().isoformat()
    try:
        table.update_item(
            Key={'LockID': resource_name},
            UpdateExpression='SET #owner = :owner, #expiry = :expiry',
            ConditionExpression=Attr('LockID').not_exists() |
 Attr('ExpiresAt').lt(now),
            ExpressionAttributeNames={'#owner': 'OwnerID', '#expiry': 'ExpiresAt'},
            ExpressionAttributeValues={':owner': owner_id, ':expiry': expiry}
        )
```

```
        return True
    except table.meta.client.exceptions.ConditionalCheckFailedException:
        return False

def release_lock(table, resource_name, owner_id):
    """Release a lock. Only succeeds if the caller is the lock owner."""
    try:
        table.delete_item(
            Key={'LockID': resource_name},
            ConditionExpression=Attr('OwnerID').eq(owner_id)
        )
        return True
    except table.meta.client.exceptions.ConditionalCheckFailedException:
        return False
```

This approach uses a condition expression to ensure that a lock can only be acquired if it doesn't exist or has expired, and can only be released by the process that acquired it. Consider enabling Time to Live (TTL) on the lock table to automatically clean up expired lock items.

Choosing a concurrency control strategy

Use the following guidelines to choose the right approach for your workload:

Use optimistic locking when:

- Conflicts are infrequent.
- Retrying a failed write is inexpensive.
- You are updating a single item at a time.

Use transactions when:

- You need to update multiple items atomically.
- You require all-or-nothing semantics across items or tables.
- You need to combine condition checks with writes in a single operation.

Use the lock client when:

- You need to coordinate access to external resources across distributed processes.
- The critical section is long-running and retrying on conflict is expensive.
- You need automatic lock expiry to handle process failures.

> **Note**
>
> If you use DynamoDB global tables, be aware that global tables use a "last writer wins" reconciliation strategy for concurrent updates. Optimistic locking with version numbers does not work as expected across Regions because a write in one Region may overwrite a concurrent write in another Region without a version check. Design your application to handle conflicts at the application level when using global tables.

Best practices for understanding your AWS billing and usage reports in DynamoDB

This document explains the `UsageType` billing codes for charges related to DynamoDB.

AWS provides cost and usage reports (CUR) that contain data for the services used. You can use AWS Cost and Usage Report to publish billing reports to Amazon S3 in a CSV format. When setting up the CUR you can choose to break time periods down by hour, day, or month, and you can choose if you want to break out usage by resource ID or not. For more details on generating CUR, please see Creating Cost and Usage Reports

Within the CSV export, you will find relevant attributes listed for each line. The following are examples of attributes that may be included:

- **lineitem/UsageStartDate:** The start date and time for the line item in UTC, inclusive.
- **lineitem/UsageEndDate:** The end date and time for the corresponding line item in UTC, exclusive.
- **lineitem/ProductCode:** For DynamoDB this will be “AmazonDynamoDB”
- **lineitem/UsageType:** A specific description code for the type of usage, as enumerated in this document
- **lineitem/Operation:** A name that provides context to the charge such as the operation name that incurred the charge (optional).
- **lineitem/ResourceId:** The identifier for the resource that incurred the usage. Available if the CUR includes a breakdown by resource ID.
- **lineitem/UsageAmount:** The amount of usage incurred during the specified time period.
- **lineitem/UnblendedCost:** The cost of this usage.
- **lineitem/LineItemDescription:** Textual description of the line item.

For more information about the CUR data dictionary, see Cost and Usage Report (CUR) 2.0. Note that the exact names vary depending on context.

A `UsageType` is a string with a value such as `ReadCapacityUnit-Hrs`, `USW2-ReadRequestUnits`, `EU-WriteCapacityUnit-Hrs`, or `USE1-TimedPITRStorage-ByteHrs`. Each usage type begins with an optional Region prefix. If absent, that indicates the us-east-1 Region. If present, the below table maps the short billing Region code to the conventional Region code and name.

For example, the usage named `USW2-ReadRequestUnits` indicates read request units consumed in us-west-2.

Billing Region Code	Region Code	Region Name
AFS1	af-south-1	Africa (Cape Town)
APE1	ap-east-1	Asia Pacific (Hong Kong)
APN1	ap-northeast-1	Asia Pacific (Tokyo)
APN2	ap-northeast-2	Asia Pacific (Seoul)
APN3	ap-northeast-3	Asia Pacific (Osaka)
APS1	ap-southeast-1	Asia Pacific (Singapore)
APS2	ap-southeast-2	Asia Pacific (Sydney)
APS3	ap-south-1	Asia Pacific (Mumbai)
APS4	ap-southeast-3	Asia Pacific (Jakarta)
APS5	ap-south-2	Asia Pacific (Hyderabad)
APS6	ap-southeast-4	Asia Pacific (Melbourne)
CAN1	ca-central-1	Canada (Central)
EU	eu-west-1	Europe (Ireland)
EUC1	eu-central-1	Europe (Frankfurt)

Billing Region Code	Region Code	Region Name
EUC2	eu-central-2	Europe (Zurich)
EUN1	eu-north-1	Europe (Stockholm)
EUS1	eu-south-1	Europe (Milan)
EUS2	eu-south-2	Europe (Spain)
EUW1	eu-west-1	Europe (Ireland)
EUW2	eu-west-2	Europe (London)
EUW3	eu-west-3	Europe (Paris)
ILC1	Il-central-1	Israel (Tel Aviv)
MEC1	me-central-1	Middle East (UAE)
MES1	me-south-1	Middle East (Bahrain)
SAE1	sa-east-1	South America (São Paulo)
USE1 (default)	us-east-1	US East (N. Virginia)
USE2	us-east-2	US East (Ohio)
UGE1	us-gov-east-1	US Government East
UGW1	us-gov-west-1	US Government West
USW1	us-west-1	US West (N. California)
USW2	us-west-2	US West (Oregon)

In the following sections, we use `REG-UsageType` pattern when going through the charges for DynamoDB, where REG specifies the region where usage occurred and usageType is the code for the type of charge. For example if you see a line item for `USW1- ReadCapacityUnit-Hrs` in your CSV file, that means the usage was incurred in US-West-1 for provisioned read capacity. In that case the listing would say `REG-ReadCapacityUnit-Hrs`.

Topics

- Throughput Capacity
- Streams
- Storage
- Backup and Restore
- Data Transfer
- CloudWatch Contributor Insights
- DynamoDB Accelerator (DAX)

Throughput Capacity

Provisioned Capacity Reads and Writes

When you create a DynamoDB table in provisioned capacity mode, you specify the read and write capacity that you expect your application to require. The usage type depends on your table class (Standard or Standard-Infrequent Access). You provision read and writes based on consumption rate per second, but the charges are priced per hour based on provisioned capacity.

UsageType	Units	Granularity	Description
REG-ReadCapacityUnit-Hrs	RCU-hours	Hour	Charges for reads in provisioned capacity mode using the Standard table class.
REG-IA-ReadCapacityUnit-Hrs	RCU-hours	Hour	Charges for reads in provisioned capacity mode using the Standard-IA table class.
REG-WriteCapacityUnit-Hrs	WCU-hours	Hour	Charges for writes in provisioned capacity mode using the Standard table class.

UsageType	Units	Granularity	Description
REG-IA-WriteCapacityUnit-Hrs	WCU-hours	Hour	Charges for writes in provisioned capacity mode using the Standard-IA table class.

Reserved Capacity Reads and Writes

With reserved capacity, you pay a one-time upfront fee and commit to a minimum provisioned usage level over a period of time. Reserved capacity is billed at a discounted hourly rate. Any capacity that you provision in excess of your reserved capacity is billed at standard provisioned capacity rates. Reserved capacity is available for single-region, provisioned read and write capacity units (RCU and WCU) on DynamoDB tables that use the standard table class. Both 1-year and 3-year reserved capacity are billed using the same SKUs.

UsageType	Units	Granularity	Description
REG-HeavyUsage:dynamodb.read	RCU-hours	Up-front then monthly	Charges for reserved capacity reads: a one-time up-front charge and a monthly charge at the start of each month covering all the discounted committed RCU-hours during the month. Will have matching zero-cost REG-ReadCapacityUnit-Hrs line items.
REG-HeavyUsage:dynamodb.write	WCU-hours	Up-front then monthly	Charges for reserved capacity writes: a one-time up-front charge and a monthly

UsageType	Units	Granularity	Description
			charge at the start of each month covering all the discounted committed WCU-hours during the month. Will have matching zero-cost REG-WriteCapacityUnit-Hrs line items.

On-Demand Capacity Reads and Writes

When you create a DynamoDB table in on-demand capacity mode, you pay only for the reads and writes your application performs. The prices for read and write requests depend on your table class.

UsageType	Units	Granularity	Description
REG-ReadRequestUnits	RRUs	Unit	Charges for reads in on-demand capacity mode with Standard table class.
REG-IA-ReadRequestUnits	RRUs	Unit	Charges for reads in on-demand capacity mode with Standard-IA table class.
REG-WriteRequestUnits	WRUs	Unit	Charges for writes in on-demand capacity mode with Standard table class.
REG-IA-WriteRequestUnits	WRUs	Unit	Charges for writes in on-demand capacity

UsageType	Units	Granularity	Description
			mode with Standard-IA table class.

Global Tables Reads and Writes

DynamoDB charges for global tables usage based on the resources used on each replica table. For provisioned global tables, write requests for global tables are measured in replicated WCUs (rWCU) instead of standard WCUs and writes to global secondary indexes in global tables are measured in WCUs. For on-demand global tables, write requests are measured in replicated WRUs (rWRU) instead of standard WRUs. The number of rWCUs or rWRUs consumed for replication depends on the version of global tables you are using. The pricing depends on your table class.

Writes to global secondary indexes (GSIs) are billed using standard write units (WCUs and WRUs). Read requests and data storage are billed identically to single-region tables.

If you add a table replica to create or extend a global table in new Regions, DynamoDB charges for a table restore in the added Regions per gigabyte of data restored. Restored Data is charged as REG-RestoreDataSize-Bytes. Please refer to Backup and restore for DynamoDB for details. Cross-Region replication and adding replicas to tables that contain data also incur charges for data transfer out.

When you select on-demand capacity mode for your DynamoDB global tables, you pay only for the resources your application uses on each replica table.

UsageType	Units	Granularity	Description
REG-ReplWriteCapacityUnit-Hrs	rWCU-hours	Hour	Global table, provisioned, Standard table class.
REG-IA-ReplWriteCapacityUnit-Hrs	rWCU-hours	Hour	Global table, provisioned, Standard-IA table class.

UsageType	Units	Granularity	Description
REG-ReplWriteRequestUnits	rWRU	Unit	Global table, on-demand, Standard table class.
REG-IA-ReplWriteRequestUnits	rWRU	Unit	Global table, on-demand, Standard- IA table class

Streams

DynamoDB has two streaming technologies, DynamoDB Streams and Kinesis. Each have separate pricing.

DynamoDB Streams charges for reading data in read request units. Each `GetRecords` API call is billed as a streams read request. You are not charged for `GetRecords` API calls invoked by AWS Lambda as part of DynamoDB triggers or by DynamoDB global tables as part of replication.

UsageType	Units	Granularity	Description
REG-Streams-RequestsCount	Count	Unit	Read request units for DynamoDB Streams.

Amazon Kinesis Data Streams charges in change data capture units. DynamoDB charges one change data capture unit for each write (up to 1 KB). For items larger than 1 KB, additional change data capture units are required. You pay only for the writes your application performs without having to manage throughput capacity on the table.

UsageType	Units	Granularity	Description
REG-ChangeDataCaptureUnits-Kinesis	CDC Units	Unit	Change data capture units for Kinesis Data Streams.

Storage

DynamoDB measures the size of your billable data by adding the raw byte size of your data plus a per-item storage overhead that depends on the features you have enabled.

> **Note**
>
> Storage usage values in the CUR will be higher compared with the storage values when using `DescribeTable`, because `DescribeTable` does not include the per-item storage overhead.

Storage is calculated hourly but priced monthly as calculated from an average of the hourly charges.

Although the storage `UsageType` uses `ByteHrs` as a suffix, storage usage in the CUR is measured in GB and priced by GB-month.

UsageType	Units	Granularity	Description
REG-TimedStorage-ByteHrs	GB	Month	Amount of storage used by your DynamoDB tables and indexes, for tables with the Standard table class.
REG-IA-TimedStorage- ByteHrs	GB	Month	Amount of storage used by your DynamoDB tables and indexes, for tables with the Standard-IA table class.

Backup and Restore

DynamoDB offers two types of backups: Point In Time Recovery (PITR) backups and on- demand backups. Users can also restore from those backups into DynamoDB tables. The charges below refers to both backups and restores.

Backup storage charges are incurred on the first of the month with adjustments made throughout the month as backups are added or removed. See the Understanding Amazon DynamoDB On-demand Backups and Billing blog for more information

UsageType	Units	Granularity	Description
REG-TimedBackupStorage-ByteHrs	GB	Month	The storage consumed by on-demand backups of your DynamoDB tables and Local Secondary Indexes.
TimedPITRStorage-ByteHrs	GB	Month	The storage used by point-in-time recovery (PITR) backups. DynamoDB monitors the size of your PITR-enabled tables continuously throughout the month to determine your backup charges and bills for storage as long as PITR is enabled.
REG-RestoreDataSize-Bytes	GB	Size	The total size of data restored (including table data, local secondary indexes and global secondary

UsageType	Units	Granularity	Description
			indexes) measured in GB from DynamoDB backups.

AWS Backup

AWS Backup is a fully managed backup service that makes it easy to centralize and automate the backup of data across AWS services in the cloud as well as on premises. AWS Backup is charged for storage (warm or cold storage), restoration activities, and cross-Region data transfer. The following UsageType charges appear under the "AWSBackup" ProductCode rather than "AmazonDynamoDB".

UsageType	Units	Granularity	Description
REG-WarmStorage-ByteHrs-DynamoDB	GB	Month	The storage used by DynamoDB backups managed by AWS Backup throughout the month, measured in GB-Month.
REG-CrossRegion-WarmBytes-DynamoDB	GB	Size	The data transferred to a different AWS Region either within the same account or to a different AWS account. Cross-Region transfers charges occur when copying backups from one Region to another Region. The charge is always billed to the account where the

UsageType	Units	Granularity	Description
			data is transferred from.
REG-Restore-WarmBytes-DynamoDB	GB	Size	The total size of the data restored from warm storage, measured in GB.
REG-ColdStorage-ByteHrs-DynamoDB	GB	Month	The cold storage used by DynamoDB backups managed by AWS Backup throughout the month, measured in GB-Month.
REG-Restore-ColdBytes-DynamoDB	GB	Month	The total size of the data restored from cold storage, measured in GB.

Export and Import

You can export data from DynamoDB to Amazon S3 or import data from Amazon S3 to a new DynamoDB table.

Although the `UsageType` uses `Bytes` as a suffix, export and import usage in the CUR is measured and priced in GB.

UsageType	Units	Granularity	Description
REG-ExportDataSize-Bytes	GB	Size	The charge for exporting data to S3. DynamoDB charges for data you export based on the size

UsageType	Units	Granularity	Description
			of the DynamoDB base table (table data and local secondary indexes) at the specified point in time when the export was created.
REG-ImportDataSize-Bytes	GB	Size	The charge for importing data from S3. The size is calculated based on the uncompressed object size of the data within Amazon S3. There are no extra charges for importing to tables with GSIs.
REG-IncrementalExportDataSize-Bytes	GB	Size	The charge for size of the data processed from the continuous backup to produce incremental exports.

Data Transfer

Data transfer activity may appear associated with the DynamoDB service. DynamoDB does not charge for inbound data transfer, and it does not charge for data transferred between DynamoDB and other AWS services within the same AWS Region (in other words, $0.00 per GB). Data transferred across AWS Regions (such as between DynamoDB in the US East [N. Virginia] Region and Amazon EC2 in the EU [Ireland] Region) is charged on both sides of the transfer.

UsageType	Units	Granularity	Description
REG-DataTransfer-In-Bytes	GB	Units	Data transferred in to DynamoDB from the internet.
REG-DataTransfer-Out-Bytes	GB	Units	Data transferred out from DynamoDB to the internet.

CloudWatch Contributor Insights

CloudWatch Contributor Insights for DynamoDB is a diagnostic tool for identifying the most frequently accessed and throttled keys in your DynamoDB table. The following `UsageType` charges appear under the “AmazonCloudWatch” ProductCode rather than “AmazonDynamoDB”.

UsageType	Units	Granularity	Description
REG-CW:ContributorEventsManaged	Events processed	Units	The amount of DynamoDB events processed. For example for a table with CloudWatch Contributor Insights enabled, anytime an item is read or written, it’s counted as one event. If the table has a sort key, it results in charges for two events.
REG-CW:ContributorRulesManaged	Rule count	Month	DynamoDB creates rules to identify most accessed items and most

UsageType	Units	Granularity	Description
			throttled keys when you enable Cloud Watch Contributor Insights. This charge is incurred for the rules added for each entity (tables and GSIs) configured for logging CloudWatch contributor insights.

DynamoDB Accelerator (DAX)

DynamoDB Accelerator (DAX) is billed by the hour based on the instance type selected for the service. The charges below refers to the DynamoDB Accelerator instances provisioned. The following UsageType charges appear under the “AmazonDAX” ProductCode rather than “AmazonDynamoDB”.

UsageType	Units	Granularity	Description
REG-NodeUsage:dax-<INSTANCETYPE>	Node-hour	Hour	The hourly usage of a particular instance type. Pricing is per node-hour consumed, from the time a node is launched until it is terminated. Each partial node-hour consumed will be billed as a full hour. DAX charges for each node in a DAX cluster. If you have a cluster with multiple

UsageType	Units	Granularity	Description
			nodes, you would see multiple line items in your billing report.

The instance type will be one of the values from the following list. For details about node types, see Nodes.

- r3.2xlarge, r4.8xlarge, or r5.8xlarge
- r3.4xlarge, r4.large, or r5.large
- r3.8xlarge, r4.xlarge, or r5.xlarge
- r3.2xlarge, r5.12xlarge, or t2.medium
- r3.4xlarge, r4.large, or r5.large
- r3.xlarge, r5.16xlarge, or t2.small
- r4.16xlarge, r5.24xlarge, or t3.medium
- r4.2xlarge, r5.2xlarge, or t3.small
- r4.4xlarge or r5.4xlarge

Migrating a DynamoDB table from one account to another

You can migrate an Amazon DynamoDB table from one account to another to implement a multi-account strategy or a backup strategy. You can also do it for testing, debugging, or compliance reasons. A common use case is copying DynamoDB tables across production, staging, test, and development environments where each environment utilizes a different AWS account.

DynamoDB offers two options for migrating tables from one AWS account to another:

- **AWS Backup for Cross-Account Backup and Restore:** AWS Backup is a fully managed backup service that enables you to centrally manage backups across multiple AWS services. With its cross-account backup and restore functionality, you can back up a DynamoDB table in one account and restore the backup to another account in the same AWS Organization.
- **DynamoDB Export and Import to Amazon S3:** Using the DynamoDB Export and Import to Amazon S3 features allows you to do a full export to an Amazon S3 bucket and then import that data into a new table in another AWS account. This approach is suitable when you need to

migrate between accounts that are not part of the same AWS Organization or if you do not want to use AWS Backup.

> **Note**
>
> Import from Amazon S3 does not support tables with Local Secondary Indexes (LSIs), but it does support Global Secondary Indexes (GSIs). For more information on LSIs and GSIs, see Improving data access with secondary indexes in DynamoDB.

Topics

- Migrate a table using AWS Backup for cross-account backup and restore
- Migrate a table using export to S3 and import from S3

Migrate a table using AWS Backup for cross-account backup and restore

Prerequisites

- Source and target AWS accounts must belong to the same organization in the AWS Organizations service
- Valid AWS Identity and Access Management (IAM) permissions to create and use AWS Backup vaults

For more information about setting up cross-account backups, see Creating backup copies across AWS accounts.

Pricing information

AWS charges for the backup (based on the table size), any data copying between AWS Regions (based on the amount of data), for the restore (based on the amount of data), and for any ongoing storage charges. To avoid ongoing charges, you can delete the backup if you don't need it after the restore.

For more information about pricing, see AWS Backup pricing .

Step 1: Enable advanced features for DynamoDB and cross-account backup

1. In both the source and target AWS account, access the AWS Management Console and open the AWS Backup console.
2. Choose the **Settings** option.
3. Under **Advanced features for Amazon DynamoDB backups**, confirm that **Advanced features** is enabled. If it isn't, choose **Enable**.
4. Under **Cross-account management**, for **Cross-account backup**, choose **Turn On**.

Step 2: Create a backup vault in the source account and target account

1. In the source AWS accounts, open the AWS Backup console.
2. Choose **Backup vaults**.
3. Choose **Create Backup vault**.
4. Copy and save the **Amazon Resource Name (ARN)** of the created backup vaults and the target AWS account.
5. You'll need the ARNs of both the source and target backup vaults when copying the DynamoDB table backup between accounts.

Step 3: Create a DynamoDB table backup in the source account

1. On the **AWS Backup Dashboard page**, choose **Create on-demand backup**.
2. In the **Settings** section, select **DynamoDB** as the **Resource type**, and then select the table name.
3. In the **Backup vault** dropdown list, select the backup vault you created in the source account.
4. Select the desired **Retention period**.
5. Choose **Create on-demand backup**.
6. Monitor the status of the backup job on the **Backup Jobs** tab of the **AWS Backup Jobs** page.

Step 4: Copy the DynamoDB table backup from the source account to the target account

1. After the backup job completes, open the AWS Backup console in the source account and choose **Backup vaults**.

2. Under **Backups**, choose the DynamoDB table backup. Choose **Actions** and then **Copy**.
3. Enter the AWS Region of the target account.
4. For **External vault ARN**, enter the ARN of the backup vault you created in the target account.
5. In the target account backup vault, enable access from a source account to allow copying backups.

Step 5: Restore the DynamoDB table backup in the target account

1. In the target AWS account, open the AWS Backup console and choose **Backup vaults**
2. Under **Backups**, select the backup you copied from the source account. Choose **Actions**, then **Restore**.
3. Enter the name for the new DynamoDB table, the encryption that this new table will have, the key you want the restore to be encrypted with, and any other options.
4. When the restore is completed, the table status will show as **Active**.

Migrate a table using export to S3 and import from S3

Prerequisites

- You must enable Point-in-Time Recovery (PITR) for your table in order to perform the export to S3. For more information, see Enable point-in-time recovery in DynamoDB.
- Valid IAM permissions to perform the export. For more information, see Requesting a table export in DynamoDB.
- Valid IAM permissions sufficient to perform the import. For more information, see Requesting a table import in DynamoDB.

Pricing information

AWS charges for PITR (based on the size of the table and how long PITR is enabled for). If you don't need PITR except for the export, you can turn it off after the export concludes. AWS also charges for requests made against S3, for storing the exported data in S3 and for importing (based on the uncompressed size of the imported data).

For more information about DynamoDB pricing, see DynamoDB pricing.

> **Note**
>
> There are limits on the size and number of objects when importing from S3 to DynamoDB. For more information, see Import quotas.

Requesting a table export to Amazon S3

1. Sign in to the AWS Management Console and open the DynamoDB console.
2. In the navigation pane on the left side of the console, choose **Exports to S3**.
3. Choose a source table and destination S3 bucket. Enter the URL of the destination account bucket using the `s3://bucketname/prefix` format. `/prefix` is an optional folder to help keep your destination bucket organized.
4. Choose **Full export**. A full export outputs the full table snapshot of your table, at the point in time you specify.
   a. Select **Current time** to export the latest full table snapshot.
   b. For **Exported file format**, choose between DynamoDB JSON and Amazon Ion. The default option is DynamoDB JSON.
5. Click the **Export** button to begin the export.
6. Small table exports should complete in a few minutes, but tables in the terabyte range can take more than an hour.

Requesting a table import from Amazon S3

1. Sign in to the AWS Management Console and open the DynamoDB console.
2. In the navigation pane on the left side of the console, choose **Import from S3**.
3. On the page that appears, select **Import from S3**.
4. Enter the Amazon S3 source URL. You can also find it by using the **Browse S3** button. The expected path is of the format `s3://bucket/prefix/AWSDynamoDB/<XXXXXXXX-XXXXXX>/data/`.
5. Specify if you are the S3 bucket owner.
6. Under **Import file compression**, select **GZIP** to match the export.
7. Under **Import file format**, select **DynamoDB JSON** to match the export.

8. Select **Next**. For **Specify table details**, choose the options for the new table that will be created to store your data.
9. Select **Next**. For **Configure table settings**, customize any additional table settings if applicable.
10. Select **Next** again to review your import options, then click **Import** to begin the import task. You'll see your new table listed under **Imports from S3** with the status **Importing**. You cannot access your table during this time. Small imports should complete in a few minutes, but tables in the terabyte range can take more than an hour.
11. After the import completes, the status shows as **Active**, and you can start using the table.

Keeping tables in sync during migration

If you can pause write operations on the source table for the duration of the migration, then the source and output should match up exactly after the migration. If you can't pause write operations, the target table would normally be a bit behind the source after the migration. To catch up the source table, you can use streaming (DynamoDB Streams or Kinesis Data Streams for DynamoDB) to replay the writes that happened in the source table since the backup or export.

You should start reading the stream records prior to the timestamp when you exported the source table to S3. For example, if the export to S3 occurred at 2:00 PM and the import to the target table was concluded at 11:00 PM, you should initiate the DynamoDB stream reading at 1:58 PM. The streaming options for change data capture table summarizes the features of each streaming model.

Using DynamoDB Streams with Lambda offers a streamlined approach for synchronizing data between the source and target DynamoDB tables. You can use a Lambda function to replay each write in the target table.

> **Note**
>
> Items are kept in the DynamoDB Streams for 24 hours, so you should plan to complete your backup and restore or export and import within that window.

Prescriptive guidance to integrate DAX with DynamoDB applications

DynamoDB Accelerator (DAX), is a DynamoDB-compatible caching service that provides fast in-memory performance for demanding applications, such as read-heavy applications. Using DAX, you can achieve response times in microseconds for accessing frequently requested data. This DynamoDB Accelerator prescriptive guide provides comprehensive insights and best practices for integrating DAX with your DynamoDB applications.

This guide provides foundational knowledge for those who are new to DAX or want to optimize their existing configurations. This guide covers various topics, for example, when to use DAX and creating a DAX cluster. It also includes practical examples and detailed explanations to help you effectively implement DAX in your projects. Finally, this guide offers advanced strategies that you need to implement to maximize DAX caching capabilities for ensuring fast and scalable applications.

Topics

- Evaluating the suitability of DAX for your use cases
- Configuring your DAX client
- Configuring your DAX cluster
- Sizing your DAX cluster
- Deploying a cluster
- Managing cluster operations
- Monitoring DAX

Evaluating the suitability of DAX for your use cases

This section explains when and why to use DAX. Using this guidance helps you to determine if integrating DAX with DynamoDB is advantageous for your application's workload patterns, performance requirements, and data consistency needs. It also covers scenarios where DAX might not be suitable, for example, write-heavy workloads and infrequently accessed data.

In this section

- When and why to choose DAX

- When not to use DAX

When and why to choose DAX

You can consider adding DAX to your application stack in several scenarios. For example, use DAX to reduce the overall latency of read requests against DynamoDB or to minimize repeated reads of the same data from a table. The following list presents examples of scenarios in which you can take advantage of integrating DAX with DynamoDB:

- **High-performance requirement**
  - **Low latency reads** – You should consider using DAX if your application requires response times in microseconds for eventually-consistent reads. DAX can also drastically reduce the response time for accessing frequently read data.
- **Read-intensive workloads**
  - **Read-heavy applications** – For applications with a high read-to-write ratio, for example, 10:1 or more, DAX results in more cache hits and less stale data. This reduces reads against a table. To avoid reading stale data from the cache if your application is write-heavy, make sure to set a lower Using time to live (TTL) in DynamoDB for the cache.
  - **Caching common queries** – If your application frequently reads the same data, for example, popular products on an e-commerce platform, DAX can serve these requests directly from its cache.
- **Bursty traffic patterns**
  - **Smoother table scaling** – DAX helps smooth out impacts of sudden traffic spikes. DAX provides a buffer to scale up DynamoDB table capacity gracefully, which reduces the risk of read throttling.
  - **Higher read throughput for each item** – DynamoDB allocates individual partitions for each item. However, a partition starts throttling reads of an item when it reaches 3,000 read capacity units (RCU). DAX lets you scale reads of a single item beyond 3,000 RCU.
- **Cost optimization**
  - **Reducing DynamoDB costs** – Reading from DAX can reduce reads sent to a DynamoDB table, which can then directly impact cost. With a high cache hit rate, the reduced table read cost can exceed a DAX cluster cost, which results in a net cost reduction.
- **Data consistency requirements**
  - **Eventual consistency** – DAX supports eventually consistent reads. This makes DAX suitable for use cases where immediate consistency isn't critical.

- **Write-through caching** – Writes that you make against DAX are write-through. Once DAX confirms that it's written an item to DynamoDB, it persists that item version in the item cache. This write-through mechanism helps maintain tighter data consistency between cache and database, but uses additional DAX cluster resources.

When not to use DAX

While DAX is powerful, it's not suitable for all scenarios. The following list presents examples of scenarios in which integrating DAX with DynamoDB is unsuitable:

- **Write-heavy workloads** – The primary advantage of DAX is speeding up reads, but writes use more DAX resources than reads. If your application is predominantly write-heavy, DAX benefits might be limited.
- **Infrequently read data** – If your application accesses data infrequently or a wide range of rarely reused data (cold data), you'll likely experience a low cache hit ratio. In this case, the overhead of maintaining the cache might not justify the performance gains.
- **Bulk reads or writes** – If your application performs more bulk writes than individual writes, you should write around DAX. In addition, for bulk reads, you should run full table scans directly against a DynamoDB table.
- **Strong consistency or transaction requirements** – DAX passes strongly consistent reads and TransactGetItems calls to a DynamoDB table. You should make these reads around the DAX cluster to avoid using cluster resources. Items read this way won't be cached; therefore, routing such items through DAX serves no purpose.
- **Simple applications with modest performance requirements** – For applications with modest performance requirements and tolerance for direct DynamoDB latency, the complexity and cost of adding DAX might not be necessary. On its own, DynamoDB handles high throughput and provides single-digit millisecond performance.
- **Complex querying needs beyond key-value access** – DAX is optimized for key-value access patterns. If your application requires complex querying capabilities with complex filtering, such as Query and Scan operations, DAX caching benefits might be limited.

  In these situations, use Amazon ElastiCache (Redis OSS) as an alternative. ElastiCache (Redis OSS) supports advanced data structures, such as, lists, sets, and hashes. It also offers features, such as pub/sub, geospatial indexes, and scripting.
- **Compliance requirements** – DAX doesn't currently offer the same compliance accreditations as as DynamoDB. For example, DAX hasn't obtained the SOC accreditation yet.

Configuring your DAX client

The DAX cluster is an instance-based cluster that can be accessed using various DAX SDKs. Each SDK provides developers with configurable options, such as requestTimeout and connections, to meet specific application requirements.

When configuring your DAX client, a crucial consideration is your client application's scale—specifically, the ratio of client instances to DAX server instances (which has a maximum of 11). Large client instance fleets can generate numerous connections to DAX server instances, potentially overwhelming them. This guide outlines best practices for DAX client configuration.

Best practices

1. **Client instances** – Implement singleton client instances to ensure instance reuse across requests. For implementation details, see the section called “Step 4: Run a sample application”.
2. **Request timeouts** – While applications often require low request timeouts to ensure minimal latency for upstream systems, setting timeouts too low can cause problems. Low timeouts may trigger frequent reconnection to server instances when DAX servers experience temporary latency spikes. When a timeout occurs, the DAX client terminates the existing server node connection and establishes a new one. Since connection establishment is resource-intensive, numerous consecutive connections can overload DAX servers. We recommend the following:
   - Maintaining default request timeout settings.
   - If lower timeouts are necessary, implement separate application threads with lower timeout values and include retry mechanisms with exponential back-off.
3. **Connection timeout** – For most applications, we recommend maintaining the default connection timeout settings.
4. **Concurrent connections** – Certain SDKs, such as JavaV2, allow adjustment of concurrent connections to the DAX server. Key considerations:
   - DAX server instances can handle up to 40,000 concurrent connections.
   - Default settings are suitable for most use cases.
   - Large client instances combined with high concurrent connections may overload servers.
   - Lower concurrent connection values reduce server overload risk.
   - Performance calculation example:
     - Assuming 1ms request latency, each connection can theoretically handle 1,000 requests/second.

- For a 3-node cluster, a single client instance connecting to all nodes can process 3,000 requests/second.
- With 10 connections, the client can handle approximately 30,000 requests/second.

  Recommendation – Begin with lower concurrent connection settings and validate through performance testing with expected production workload patterns.

Configuring your DAX cluster

The DAX cluster is a managed cluster, but you can adjust its configurations to fit your application requirements. Because of its close integration with DynamoDB API operations, you should consider the following aspects when integrating your application with DAX.

In this section

- DAX pricing
- Item cache and query cache
- Selecting TTL setting for the caches
- Caching multiple tables with a DAX cluster
- Data replication in DAX and DynamoDB global tables
- DAX Region availability
- DAX caching behavior

DAX pricing

The cost of a cluster depends on the number and size of nodes it has provisioned. Every node is billed for each hour it runs in the cluster. For more information, see Amazon DynamoDB pricing.

Cache hits don't incur DynamoDB cost, but impact DAX cluster resources. Cache misses incur DynamoDB read costs and require DAX resources. Writes incur DynamoDB write costs and impact DAX cluster resources to proxy the write.

Item cache and query cache

DAX maintains an item cache and a query cache. Understanding the differences between these caches can help you determine the performance and consistency characteristics they offer to your application.

Cache characteristic	Item cache	Query cache
Purpose	Stores the results of GetItem and BatchGetItem API operations.	Stores the results of Query and Scan API operations. These operations can return multiple items based on query conditions instead of specific item keys.
Access Type	Uses key-based access. When an application requests data using `GetItem` or `BatchGetItem` , DAX first checks the item cache using the primary key of the requested items. If the item is cached and unexpired , DAX returns it immediate ly without accessing the DynamoDB table.	Uses parameter-based access. DAX caches the result set of `Query` and `Scan` API operations. DAX serves subsequent requests with the same parameters that include the same query conditions, table, index, from the cache. This significantly reduces response times and DynamoDB read throughput consumption.
Cache Invalidation	DAX automatically replicates updated items into the item cache of the nodes in the DAX cluster in the following scenarios: • You write an item update through the cache. • Read an updated item version from the table.	The query cache is more challenging to invalidate than the item cache. Item updates might not directly map to cached queries or scans. You must carefully tune the query cache TTL to maintain data consistency. Writes through DAX or base table aren't reflected in query cache until the TTL expires the previousl y cached response and DAX performs a new query against DynamoDB.

Cache characteristic	Item cache	Query cache
Global secondary index	Because the `GetItem` API operation isn't supported on local secondary indexes or global secondary indexes, the item cache only caches reads from the base table.	Query cache caches queries against both tables and indexes.

Selecting TTL setting for the caches

TTL determines the period for which data is stored in the cache before it becomes stale. After this period, the data is automatically refreshed on the next request. Choosing the right TTL setting for your DAX caches involves balancing between the optimization of application performance and consistency of data. Because there doesn't exist a universal TTL setting that works for all applications, the optimal TTL setting varies based on your application's specific characteristics and requirements. We recommend that you start with a conservative TTL setting using this prescriptive guidance. Then, iteratively adjust your TTL setting based on your application's performance data and insights.

DAX maintains a least recently used (LRU) list for the item cache. The LRU list tracks when items are first written to or last read from the cache. When the DAX node memory is full, DAX evicts older items even if they haven't expired yet to make room for new items. The LRU algorithm is always enabled and not user-configurable.

To set a TTL duration that works for your applications, consider the following points:

Understand your data access patterns

- **Read-heavy workloads** – For applications with read-heavy workloads and infrequent data updates, set a longer TTL duration to reduce the number of cache misses. A longer TTL duration also reduces the need to access the underlying DynamoDB table.
- **Write-heavy workloads** – For applications with frequent updates that aren't written through DAX, set a shorter TTL duration to ensure the cache stays consistent with the database. A shorter TTL duration also reduces the risk of serving stale data.

Evaluate your application's performance requirements

- **Latency sensitivity** – If your application requires low latency over data freshness, use a longer TTL duration. A longer TTL duration maximizes cache hits, which reduces average read latency.
- **Throughput and scalability** – A longer TTL duration reduces load on DynamoDB tables and improves throughput and scalability. However, you should balance this with the need for up-to-date data.

Analyze cache eviction and memory usage

- **Cache memory limits** – Monitor your DAX cluster's memory usage. A longer TTL duration can store more data in the cache, which might reach memory limits and lead to LRU-based evictions.

Use metrics and monitoring to adjust TTL

Regularly review metrics, for example, cache hits and misses, and CPU and memory utilization. Adjust your TTL setting based on these metrics to find an optimal balance between performance and data freshness. If cache misses are high and memory utilization is low, increase the TTL duration to increase the cache hit rate.

Consider business requirements and compliance

Data retention policies might dictate the maximum TTL duration you can set for sensitive or personal information.

Cache behavior if you set TTL to zero

If you set TTL to 0, the item cache and query cache present the following behaviors:

- **Item cache** – Items in the cache are refeshed only when an LRU eviction or a write-through operation occurs.
- **Query cache** – Query responses aren't cached.

Caching multiple tables with a DAX cluster

For workloads with multiple small DynamoDB tables that don't need individual caches, a single DAX cluster caches requests for these tables. This provides more flexible and efficient use of DAX, particularly for applications that access multiple tables and require high-performance reads.

Similar to the DynamoDB data plane APIs, DAX requests require a table name. If you use multiple tables in the same DAX cluster, you don't need any specific configuration. However, you must ensure that the cluster's security permissions allow access to all cached tables.

Considerations for using DAX with multiple tables

When you use DAX with multiple DynamoDB tables, you should consider the following points:

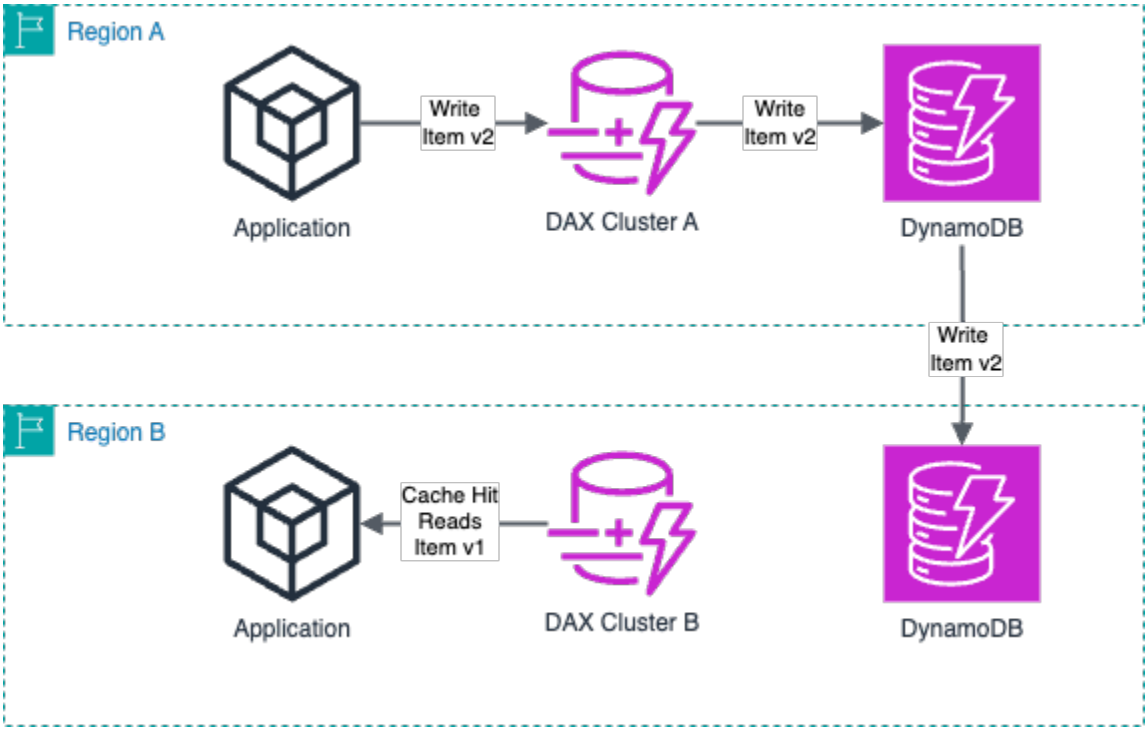
- **Memory management** – When you use DAX with multiple tables, you should consider the total size of your working data set. All the tables in your data set will share the same memory space of the node type you selected.
- **Resource allocation** – The DAX cluster's resources are shared among all the cached tables. However, a high-traffic table can cause eviction of data from the neighboring smaller tables.
- **Economies of scale** – Group smaller resources into a larger DAX cluster for averaging out traffic to a steadier pattern. For the total number of read resources that the DAX cluster requires, it's also economical to have three or more nodes. This also increases the availability of all the cached tables in the cluster.

Data replication in DAX and DynamoDB global tables

DAX is a Region-based service, so a cluster is only aware of the traffic within its AWS Region. Global tables write around the cache when they replicate data from another Region.

A longer TTL duration can cause stale data to remain in your secondary Region for longer than in the primary Region. This can result in cache misses in the local cache of the secondary Region.

The following diagram shows data replication occurring at the global table level in the source Region A. The DAX cluster in Region B isn't immediately aware of the newly replicated data from the source Region A.



DAX Region availability

Not all Regions that support DynamoDB tables support deploying DAX clusters. If your application requires low read latency through DAX, first review the list of Regions that support DAX. Then, select a Region for your DynamoDB table.

DAX caching behavior

DAX performs metadata and negative caching. Understanding these caching behaviors will help you effectively manage attribute metadata of cached items and negative cache entries.

- **Metadata caching** – DAX clusters indefinitely maintain metadata about the attribute names of cached items. This metadata persists even after the item expires or is evicted from the cache.

  Over time, applications that use unbounded number of attribute names can cause memory exhaustion in the DAX cluster. This limitation applies only to top-level attribute names, but not to the nested attribute names. Examples of unbounded attribute names include timestamps, UUIDs, and session IDs. Although you can use timestamps and session IDs as attribute values, we recommend to use shorter and more predictable attribute names.
- **Negative caching** – If a cache miss occurs and the read from a DynamoDB table yields no matching items, DAX adds a negative cache entry in the respective item or query cache. This

entry remains until the cache TTL duration expires or a write-through occurs. DAX continues to return this negative cache entry for future requests.

If the negative caching behavior doesn't fit your application pattern, read the DynamoDB table directly when DAX returns an empty result. We also recommend that you set a lower TTL cache duration to avoid long-lasting empty results in the cache and improve consistency with the table.

Sizing your DAX cluster

A DAX cluster's total capacity and availability depends on node type and count. More nodes in the cluster increase its read capacity, but not the write capacity. Larger node types (up to r5.8xlarge) can handle more writes, but too few nodes can impact availability when a node failure occurs. For more information about sizing your DAX cluster, see the DAX cluster sizing guide.

The following sections discuss the different sizing aspects that you should consider to balance node type and count for creating a scalable and cost-efficient cluster.

In this section

- Planning availability
- Planning write throughput
- Planning read throughput
- Planning dataset size
- Calculating approximate cluster capacity requirements
- Approximating cluster throughput capacity by node type
- Scaling write capacity in DAX clusters

Planning availability

When sizing a DAX cluster, you should first focus on its targeted availability. Availability of a clustered service, such as DAX, is a dimension of the total number of nodes in the cluster. Because a single node cluster has no tolerance for failure, its availability is equal to one node. In a 10-node cluster, the loss of a single node has a minimal impact to the cluster's overall capacity. This loss doesn't have a direct impact on availability because the remaining nodes can still fulfill read requests. To resume writes, DAX quickly nominates a new primary node.

DAX is VPC-based. It uses a subnet group to determine which Availability Zones it can run nodes in and which IP addresses to use from the subnets. For production workloads, we highly recommend that you use DAX with at least three nodes in different Availability Zones. This ensures that the cluster has more than one node left to handle requests even if a single node or Availability Zone fails. A cluster can have up to 11 nodes, where one is a primary node and 10 are read replicas.

Planning write throughput

All DAX clusters have a primary node for write-through requests. The size of the node type for the cluster determines its write capacity. Adding additional read replicas doesn't increase the cluster's write capacity. Therefore, you should consider the write capacity during cluster creation because you can't change the node type later.

If your application needs to write-through DAX to update the item cache, consider increased use of cluster resources to facilitate the write. Writes against DAX consume about 25 times more resources than cache-hit reads. This might require a larger node type than for read-only clusters.

For additional guidance about determining whether write-through or write-around will work best for your application, see Strategies for writes.

Planning read throughput

A DAX cluster's read capacity depends on the cache hit ratio of your workload. Because DAX reads data from DynamoDB when a cache miss occurs, it consumes approximately 10 times more cluster resources than a cache-hit. To increase cache hits, increase the TTL setting of the cache to define the period for which an item is stored in the cache. A higher TTL duration, however, increases the chance of reading older item versions unless updates are written through DAX.

To make sure that the cluster has sufficient read capacity, scale the cluster horizontally as mentioned in Scaling a cluster horizontally. Adding more nodes adds read replicas to the pool of resources, while removing nodes reduces read capacity. When you select the number of nodes and their sizes for a cluster, consider both the minimum and maximum amount of read capacity needed. If you can't horizontally scale a cluster with smaller node types to meet your read requirements, use a larger node type.

Planning dataset size

Each available node type has a set memory size for DAX to cache data. If a node type is too small, the working set of data that an application requests won't fit in memory and results in cache

misses. Because larger nodes support larger caches, use a node type larger than the estimated data set that you need to cache. A larger cache also improves the cache hit ratio.

You might get diminishing returns for caching items with few repeated reads. Calculate the memory size for frequently accessed items and make sure the cache is large enough to store that data set.

Calculating approximate cluster capacity requirements

You can estimate your workload's total capacity needs to help you select the appropriate size and quantity of cluster nodes. To do this estimation, calculate the variable *normalized request per second* (Normalized RPS). This variable represents the total units of work your application requires the DAX cluster to support, including cache hits, cache misses, and writes. To calculate the Normalized RPS, use the following inputs:

- `ReadRPS_CacheHit` – Specifies the number of reads per second that result in a cache hit.
- `ReadRPS_CacheMiss` – Specifies the number of reads per second that result in a cache miss.
- `WriteRPS` – Specifies the number of writes per second that will go through DAX.
- `DaxNodeCount` – Specifies the number of nodes in the DAX cluster.
- `Size` – Specifies the size of the item being written or read in KB rounded up to the nearest KB.
- `10x_ReadMissFactor` – Represents a value of 10. When a cache miss occurs, DAX uses approximately 10 times more resources than cache hits.
- `25x_WriteFactor` – Represents a value of 25 because a DAX write-through uses approximately 25 times more resources than cache hits.

Using the following formula, you can calculate the Normalized RPS.

```
Normalized RPS = (ReadRPS_CacheHit * Size) + (ReadRPS_CacheMiss * Size *
 10x_ReadMissFactor) + (WriteRequestRate * 25x_WriteFactor * Size * DaxNodeCount)
```

For example, consider the following input values:

- `ReadRPS_CacheHit` = 50,000
- `ReadRPS_CacheMiss` = 1,000
- `ReadMissFactor` = 1
- `Size` = 2 KB

- `WriteRPS` = 100
- `WriteFactor` = 1
- `DaxNodeCount` = 3

By substituting these values in the formula, you can calculate the Normalized RPS as follows.

```
Normalized RPS = (50,000 Cache Hits/Sec * 2KB) + (1,000 Cache Misses/Sec * 2KB * 10) +
 (100 Writes/Sec * 25 * 2KB * 3)
```

In this example, the calculated value of Normalized RPS is 135,000. However, this Normalized RPS value doesn't account for keeping cluster utilization below 100% or node loss. We recommend that you factor in additional capacity. To do this, determine the greater of two multiplying factors: target utilization or node loss tolerance. Then, multiply the Normalized RPS by the greater factor to obtain a *target request per second* (Target RPS).

- **Target utilization**

  Because performance impacts increase cache misses, we don't recommend running the DAX cluster at 100% utilization. Ideally, you should keep cluster utilization at or below 70%. To achieve this, multiply the Normalized RPS by 1.43.

- **Node loss tolerance**

  If a node fails, your application must be able to distribute its requests among the remaining nodes. To make sure nodes stay below 100% utilization, choose a node type large enough to absorb extra traffic until the failed node comes back online. For a cluster with fewer nodes, each node must tolerate larger traffic increases when one node fails.

  If a primary node fails, DAX automatically fails over to a read replica and designates it as the new primary. If a replica node fails, other nodes in the DAX cluster can still serve requests until the failed node is recovered.

  For example, a 3-node DAX cluster with a node failure requires an additional 50% capacity on the remaining two nodes. This requires a multiplying factor of 1.5. Conversely, an 11-node cluster with a failed node requires an additional 10% capacity on the remaining nodes or a multiplying factor of 1.1.

Using the following formula, you can calculate the Target RPS.

```
Target RPS = Normalized RPS * CEILING(TargetUtilization, NodeLossTolerance)
```

For example, to calculate Target RPS, consider the following values:

- `Normalized RPS` = 135,000
- `TargetUtilization` = 1.43

  Because we're aiming for a maximum cluster utilization of 70%, we're setting `TargetUtilization` to 1.43.
- `NodeLossTolerance` = 1.5

  Say that we're using a 3-node cluster, therefore, we're setting `NodeLossTolerance` to 50% capacity.

By substituting these values in the formula, you can calculate the Target RPS as follows.

```
Target RPS = 135,000 * CEILING(1.43, 1.5)
```

In this example, because the value of `NodeLossTolerance` is greater than `TargetUtilization`, we calculate the value of Target RPS with `NodeLossTolerance`. This gives us a Target RPS of 202,500, which is the total amount of capacity the DAX cluster must support. To determine the number of nodes you'll need in a cluster, map the Target RPS to an appropriate column in the following table. For this example of a Target RPS of 202,500, you need the dax.r5.large node type with three nodes.

Approximating cluster throughput capacity by node type

Using the Target RPS formula, you can estimate cluster capacity for different node types. The following table shows approximate capacities for clusters with 1, 3, 5, and 11 node types. These capacities don't replace the need to perform load testing of DAX with your own data and request patterns. Additionally, these capacities don't include t-type instances because of their lack of fixed CPU capacity. The unit for all values in the following table is Normalized RPS.

Node type (memory)	1 node	3 nodes	5 nodes	11 nodes
dax.r5.24xlarge (768GB)	1M	3M	5M	11M
dax.r5.16xlarge (512GB)	1M	3M	5M	11M
dax.r5.12xlarge (384GB)	1M	3M	5M	11M
dax.r5.8xlarge (256GB)	1M	3M	5M	11M
dax.r5.4xlarge (128GB)	600k	1.8M	3M	6.6M
dax.r5.2xlarge (64GB)	300k	900k	1.5M	3.3M
dax.r5.xlarge (32GB)	150k	450k	750k	1.65M
dax.r5.large (16GB)	75k	225k	375k	825k

Because of the maximum limit of 1 million NPS (network operations per second) for each node, nodes of types dax.r5.8xlarge or larger don't contribute additional cluster capacity. Node types larger than 8xlarge might not contribute to total throughput capacity of the cluster. However, such node types can be helpful for storing a larger working data set in memory.

Scaling write capacity in DAX clusters

Each write to DAX consumes 25 normalized requests on every node. Because there's a 1 million RPS limit for each node, a DAX cluster is limited to 40,000 writes per second, not accounting for read usage.

If your use case requires more than 40,000 writes per second in the cache, you must use separate DAX clusters and shard the writes among them. Similar to DynamoDB, you can hash the partition key for the data you're writing to the cache. Then, use modulus to determine which shard to write the data to.

The following example calculates the hash of an input string. It then calculates the modulus of the hash value with 10.

```
def hash_modulo(input_string):
    # Compute the hash of the input string
    hash_value = hash(input_string)

    # Compute the modulus of the hash value with 10
    bucket_number = hash_value % 10

    return bucket_number

#Example usage
if _name_ == "_main_":
    input_string = input("Enter a string: ")
    result = hash_modulo(input_string)
    print(f"The hash modulo 10 of '{input_string}' is: {result}.")
```

Deploying a cluster

Creating a new DAX cluster requires configurations beyond those needed for DynamoDB. These configurations are particularly for networking because DAX is based on Amazon VPC. This gives you complete control over your virtual networking environment, including resource placement, connectivity, and security. This section presents the best practices for the settings needed during cluster creation.

For information about choosing cluster nodes, see Sizing your DAX cluster.

In this section

- Configure networks
- Configure security
- Parameter group
- Maintenance window

Configure networks

DAX uses a subnet group to determine which Availability Zones it can run nodes in and which IP addresses to use from the subnets. To minimize latency between your application and DAX, the subnets and Availability Zones for your application servers and the DAX cluster should be the same.

We recommend that you spread the DAX nodes across multiple Availability Zones. The default option of Automatic allocation does this for you.

For best practices about setting up your VPC, see Get started with Amazon VPC in the *Amazon VPC User Guide*.

Configure security

This section discusses the security measures that you should implement for your applications that use DAX. This section also briefly discusses the support that DAX includes for data encryption.

IAM

DAX and DynamoDB have separate access control mechanisms. DAX requires an IAM role to access your DynamoDB tables. This role should follow the principle of least privilege and grant access only to specific tables and DynamoDB operations, such as GetItem and PutItem. For more information about the access control mechanisms provided by DAX, see DAX access control.

Encryption

You configure encryption at rest and encryption in transit while creating a DAX cluster. These are enabled by default. We recommend that you keep the default encryption settings unless business requirements prevent it. For more information, see DAX encryption at rest and DAX encryption in transit.

Parameter group

DAX applies a set of configurations on every node in a cluster called a parameter group. You can change this configuration after creating the cluster.

The DAX parameter group holds TTL settings for item cache and query cache. By default, the TTL duration is 5 minutes. You can override the TTL duration to any integer value greater than or equal to 1 millisecond.

You can't modify parameter groups when a running DAX instance is using them. You can change the parameter group values during the downtime of a DAX cluster.

Maintenance window

To allow for occasional software upgrades and patches to your nodes, a weekly maintenance window is configured for the DAX cluster. During this window, DAX performs rolling updates to the nodes. Clusters with more than one node don't lose availability of the cluster during these updates, but have reduced cluster capacity until the node returns. If your organization has a predictable time of low usage, consider setting the maintenance window manually to this time.

Managing cluster operations

DAX handles the cluster's maintenance and health for you. However, you need to provide operational input to scale the cluster horizontally or vertically to match your usage patterns. This section describes the recommended process to scale your DAX clusters.

In this section

- Scaling a cluster horizontally
- Scaling a cluster vertically

Scaling a cluster horizontally

Scaling a DAX cluster involves adjusting its capacity to meet throughput demands. This adjustment is done by increasing or decreasing the number of nodes (replicas) in the cluster while it's running. This process, known as horizontal scaling, helps distribute the workload across more nodes or consolidate to fewer nodes when demand is low.

You can horizontally scale your DAX cluster in and out using the `decrease-replication-factor` or `increase-replication-factor` commands in the AWS CLI.

Increase replication factor (scale out)

Increasing the replication factor of a DAX cluster adds more nodes to the cluster. The following example shows the usage of the `increase-replication-factor` command.

```
aws dax increase-replication-factor \
    --cluster-name yourClusterName  \
```

```
    --new-replication-factor desiredReplicationFactor
```

- In this command, the `cluster-name` argument specifies the name of your cluster. For example, *yourClusterName*.
- The `new-replication-factor` argument specifies the total number of nodes to add in the cluster after scaling. This includes the primary node and replica nodes. For example, if your cluster currently has 3 nodes and you want to add 2 more nodes, set the value of `new-replication-factor` to 5.

Decrease replication factor (scale in)

Decreasing the replication factor of a DAX cluster removes nodes from the cluster. Removing nodes can help reduce cost during periods of low demand. The following example shows the usage of the `decrease-replication-factor` command.

```
aws dax decrease-replication-factor \
    --cluster-name yourClusterName  \
    --new-replication-factor desiredReplicationFactor
```

- In this command, the `cluster-name` argument specifies the name of your cluster. For example, *yourClusterName*.
- The `new-replication-factor` argument specifies the reduced number of nodes in your cluster after scaling. This number must be lower than the current replication factor and must include the primary node. For instance, if your cluster has 5 nodes and you want to remove 2 nodes, set the value of `new-replication-factor` to 3.

Horizontal scaling considerations

Consider the following when you plan horizontal scaling:

- **Primary node** – The DAX cluster includes a primary node. The replication factor includes this primary node. For example, a replication factor of 3 means one primary node and two replica nodes.
- **Availability** – Adding or removing DAX nodes changes the cluster's availability and fault tolerance. More nodes can improve availability, but they also increase costs.
- **Data migration** – When you increase the replication factor, DAX automatically handles data distribution across the new set of nodes. When a new node begins serving traffic, its cache

is already warmed. However, during this process, there might be a temporary impact on performance during data migration.

Make sure you monitor your DAX clusters closely during and after the scaling process to ensure they're performing as expected and make further adjustments as necessary.

Scaling a cluster vertically

To vertically scale the node size of an existing cluster, you need to create a new cluster and migrate the application traffic to the new cluster. Migrating to a new cluster with different nodes involves several steps to ensure a smooth transition with minimal impact on your application's performance and availability.

To create a new cluster for scaling your node size vertically, consider the following points:

- **Access your current setup** – Review the metrics of your current DAX cluster to determine the new node size and quantity you need. Use this information as input to define your cluster size. For information, see Sizing your DAX cluster.
- **Set up a new DAX cluster** – Create a new DAX cluster with the node type and quantity you determined. You can use the existing configuration settings from your parameter group, unless you need to make adjustments.
- **Synchronize data** – Because DAX is a caching layer for DynamoDB, you don't need to migrate data directly. However, the new DAX cluster won't have any of your working dataset in memory until you send traffic to it.
- **Update application configuration** – Update your application's configuration to point to the new DAX cluster's endpoint. You might need to change code or update environment variables, depending on your application's configuration.

  To reduce impact when you switch to a new cluster, send canary traffic to the new cluster from a small portion of your application fleet. You can do this by slowly rolling out application updates or by using a weight-based routing DNS entry in front of your DAX endpoint.
- **Monitor and optimize** – After you switch to the new DAX cluster, closely monitor its performance metrics and logs for any issues. Be ready to adjust the number of nodes based on updated workload patterns.

  Until the new cluster caches your working dataset properly, you'll see higher cache miss rates and latencies.

- **Decommission old cluster** – When you're sure that the new cluster is performing as expected, safely decommission the old DAX cluster to avoid unnecessary costs.

Monitoring DAX

You can monitor key metrics, for example cache hit ratio, to ensure optimal DAX cluster performance, diagnose issues, and determine when you need to scale the cluster. Regularly checking key metrics helps you maintain performance, stability, and cost-efficiency by scaling the cluster to match your workload requirements. For more information about monitoring DAX, see Production monitoring.

The following list presents some of the key metrics you should monitor:

- **Cache hit ratio** – Shows how effectively DAX serves cached data, reducing the need to access the underlying DynamoDB tables. Few cache misses for the cluster indicate good caching efficiency. But few cache hits suggest that you might need to revisit the caching TTL setting or the workload isn't a good fit for caching.

  Use Amazon CloudWatch to calculate your DAX cluster's cache hit ratio. Compare the `ItemCacheHits`, `ItemCacheMisses`, `QueryCacheHits`, and `QueryCacheMisses` metrics to get this ratio. The following formula shows how the cache hit ratio is calculated. To calculate the ratio using this formula, divide your cache hits by the sum of your cache hits and misses.

  ```
  Cache hit ratio = Cache hits / (Cache hits + Cache misses)
  ```

  The cache hit ratio is a number between 0 and 1, which is represented as a percentage. A higher percentage indicates better overall cache utilization.
- **ErrorRequestCount** – Count of requests that resulted in user errors reported by the node or cluster. `ErrorRequestCount` includes requests that were throttled by the node or cluster. Monitoring user errors can help you identify scaling misconfigurations or hot item/partition patterns in your application.
- **Operation latencies** – Monitoring the latency of read and write operations to and from the DAX cluster can help you in identifying performance bottlenecks. Increasing latencies might indicate issues with your DAX cluster configuration, network, or the need to scale.
- **Network consumption** – Keep an eye on the `NetworkBytesIn` and `NetworkBytesOut` metrics to monitor your DAX cluster's network traffic. An unexpected increase in network throughput

could mean more client requests or inefficient query patterns that are causing more data to be transferred.

Monitoring network consumption helps you manage costs for your DAX cluster. It also ensures the network doesn't become a bottleneck for cluster performance.

- **Eviction rate** – Shows how often items are removed from your cache to make room for new items. If the eviction rate increases over time, your cache might be too small or your caching strategy isn't effective.

  Monitor the `EvictedSize` metric in CloudWatch to determine if your cache size is adequate for your workload. If the total evicted size keeps growing, you might need to scale up your DAX cluster to accommodate a larger cache.

- **CPU utilization** – Refers to the percentage of CPU utilization of the node or cluster. This is a critical metric to monitor for any database or caching system. High CPU utilization could mean your DAX cluster might be overloaded and needs scaling to handle the increased demand.

  Monitor the `CPUUtilization` metric for your DAX cluster. If your CPU utilization consistently approaches or exceeds 70-80%, consider scaling up your DAX cluster as described in the following section.

  If the number of requests sent to DAX exceeds a node's capacity, DAX limits the rate at which it accepts additional requests. It does this by returning a ThrottlingException. DAX continuously evaluates your cluster's CPU utilization to determine the request volume it can process while maintaining a healthy cluster state.

  You can monitor the `ThrottledRequestCount` metric that DAX publishes to CloudWatch. If you see these exceptions regularly, you should consider scaling up your cluster.

Scaling your DAX cluster using monitoring data

You can determine if you need to scale up or down your DAX cluster by monitoring its performance metrics.

- **Scale up or out** – If your DAX cluster has high CPU utilization, low cache hits (after optimizing the caching strategy), or high operation latencies, you should scale up your cluster. Adding more nodes, also called scaling out, can help distribute the load more evenly. For workloads with increasing writes per second, you might need to choose more powerful nodes (scaling up).

- **Scale down** – If you consistently see low CPU utilization and operation latencies below your thresholds, you might have over-provisioned resources. In such cases, scale down nodes to reduce costs. You can reduce the number of nodes down to 1 during low utilization periods, but you can't shut the cluster down entirely.

Using DynamoDB with other AWS services

Amazon DynamoDB is integrated with other AWS services, letting you automate repeating tasks or build applications that span multiple services.

Topics

- Configuring AWS credentials using Amazon Cognito for DynamoDB
- Integrating with Amazon Redshift
- Processing DynamoDB data with Apache Hive on Amazon EMR
- Integrating DynamoDB with Amazon S3
- DynamoDB zero-ETL integration with Amazon SageMaker Lakehouse
- DynamoDB zero-ETL integration with Amazon OpenSearch Service
- Integrating DynamoDB with Amazon EventBridge
- Integrating DynamoDB with Amazon Managed Streaming for Apache Kafka
- Best practices for integrating with DynamoDB

Configuring AWS credentials using Amazon Cognito for DynamoDB

The recommended way to obtain AWS credentials for your web and mobile applications is to use Amazon Cognito. Amazon Cognito helps you avoid hardcoding your AWS credentials on your files. It uses AWS Identity and Access Management (IAM) roles to generate temporary credentials for your application's authenticated and unauthenticated users.

For example, to configure your JavaScript files to use an Amazon Cognito unauthenticated role to access the Amazon DynamoDB web service, do the following.

To configure credentials to integrate with Amazon Cognito

1. Create an Amazon Cognito identity pool that allows unauthenticated identities.

```
aws cognito-identity create-identity-pool \
    --identity-pool-name DynamoPool \
    --allow-unauthenticated-identities \
    --output json
```

```
{
    "IdentityPoolId": "us-west-2:12345678-1ab2-123a-1234-a12345ab12",
    "AllowUnauthenticatedIdentities": true,
    "IdentityPoolName": "DynamoPool"
}
```

2. Copy the following policy into a file named `myCognitoPolicy.json`. Replace the identity pool ID (*us-west-2:12345678-1ab2-123a-1234-a12345ab12*) with your own `IdentityPoolId` obtained in the previous step.

   JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Federated": "cognito-identity.amazonaws.com"
      },
      "Action": "sts:AssumeRoleWithWebIdentity",
      "Condition": {
        "StringEquals": {
          "cognito-identity.amazonaws.com:aud": "us-
west-2:12345678-1ab2-123a-1234-a12345ab12"
        },
        "ForAnyValue:StringLike": {
          "cognito-identity.amazonaws.com:amr": "unauthenticated"
        }
      }
    }
  ]
}
```

3. Create an IAM role that assumes the previous policy. In this way, Amazon Cognito becomes a trusted entity that can assume the `Cognito_DynamoPoolUnauth` role.

```
aws iam create-role --role-name Cognito_DynamoPoolUnauth \
--assume-role-policy-document file://PathToFile/myCognitoPolicy.json --output json
```

4. Grant the `Cognito_DynamoPoolUnauth` role full access to DynamoDB by attaching a managed policy (`AmazonDynamoDBFullAccess`).

```
aws iam attach-role-policy --policy-arn arn:aws:iam::aws:policy/
AmazonDynamoDBFullAccess \
--role-name Cognito_DynamoPoolUnauth
```

> **Note**
>
> Alternatively, you can grant fine-grained access to DynamoDB. For more information, see Using IAM policy conditions for fine-grained access control.

5. Obtain and copy the IAM role Amazon Resource Name (ARN).

```
aws iam get-role --role-name Cognito_DynamoPoolUnauth --output json
```

6. Add the `Cognito_DynamoPoolUnauth` role to the `DynamoPool` identity pool. The format to specify is `KeyName=string`, where `KeyName` is `unauthenticated` and the string is the role ARN obtained in the previous step.

```
aws cognito-identity set-identity-pool-roles \
--identity-pool-id "us-west-2:12345678-1ab2-123a-1234-a12345ab12" \
--roles unauthenticated=arn:aws:iam::123456789012:role/Cognito_DynamoPoolUnauth --
output json
```

7. Specify the Amazon Cognito credentials in your files. Modify the `IdentityPoolId` and `RoleArn` accordingly.

```
AWS.config.credentials = new AWS.CognitoIdentityCredentials({
IdentityPoolId: "us-west-2:12345678-1ab2-123a-1234-a12345ab12",
RoleArn: "arn:aws:iam::123456789012:role/Cognito_DynamoPoolUnauth"
});
```

You can now run your JavaScript programs against the DynamoDB web service using Amazon Cognito credentials. For more information, see Setting credentials in a web browser in the *AWS SDK for JavaScript Getting Started Guide*.

Integrating with Amazon Redshift

Amazon Redshift is a fast, fully managed, petabyte-scale data warehouse service that makes it simple and cost-effective to efficiently analyze all your data using your existing business intelligence tools.

DynamoDB and Amazon Redshift can be used together to address different data storage and processing needs within an application or data ecosystem.

See the topics below for more detailed topics on how to integrate DynamoDB with Amazon Redshift.

Topics

- Cross-account integration considerations with CMK
- DynamoDB zero-ETL integration with Amazon Redshift
- Loading data from DynamoDB into Amazon Redshift with the COPY command

Cross-account integration considerations with CMK

When you attempt to integrate from DynamoDB to Amazon Redshift, the initial action is launched from Amazon Redshift. Without the proper permissions, this action could result in a silent failure. The following sections detail the permissions required for this cross-account integration.

Required AWS KMS policies and permissions

Replace the following placeholders in the examples:

- `111122223333`: The AWS account ID where Amazon Redshift is hosted
- `444455556666`: The AWS account ID where DynamoDB is hosted
- `REDSHIFT_ROLE_NAME`: The IAM role name used by Amazon Redshift
- `REGION`: The AWS Region where your resources are located
- `TABLE_NAME`: The name of your DynamoDB table
- `KMS_KEY_ID`: The ID of your KMS key

KMS key policy in the DynamoDB account

The following AWS KMS key policy enables cross-account access between your DynamoDB and Amazon Redshift services. In this example, account 444455556666 contains the DynamoDB table and AWS KMS key, while account 111122223333 contains the Amazon Redshift cluster that needs access to decrypt the data.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "Enable IAM User Permissions",
            "Effect": "Allow",
            "Principal": {
                "AWS": "arn:aws:iam::444455556666:root"
            },
            "Action": "kms:*",
            "Resource": "*"
        },
        {
            "Sid": "Allow Redshift to use the key",
            "Effect": "Allow",
            "Principal": {
                "AWS": "arn:aws:iam::111122223333:role/REDSHIFT_ROLE_NAME"
            },
            "Action": [
                "kms:Decrypt",
                "kms:DescribeKey",
                "kms:GenerateDataKey",
                "kms:GenerateDataKeyWithoutPlaintext"
            ],
            "Resource": "*"
        }
    ]
}
```

IAM Policy for the Amazon Redshift role (in Amazon Redshift account)

The following IAM policy allows a Amazon Redshift service to access DynamoDB tables and their associated AWS KMS encryption keys in a cross-account scenario. In this example, account 444455556666 contains the DynamoDB resources and AWS KMS keys that the Amazon Redshift service needs to access.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "AllowDynamoDBAccess",
            "Effect": "Allow",
            "Action": [
                "dynamodb:DescribeTable",
                "dynamodb:BatchGetItem",
                "dynamodb:Scan",
                "dynamodb:Query",
                "dynamodb:BatchGetItem",
                "dynamodb:GetItem",
                "dynamodb:GetRecords",
                "dynamodb:GetShardIterator",
                "dynamodb:DescribeStream",
                "dynamodb:ListStreams"
            ],
            "Resource": [
                "arn:aws:dynamodb:*:444455556666:table/TABLE_NAME",
                "arn:aws:dynamodb:*:444455556666:table/TABLE_NAME/stream/*"
            ]
        },
        {
            "Sid": "AllowKMSAccess",
            "Effect": "Allow",
            "Action": [
                "kms:Decrypt",
                "kms:DescribeKey",
                "kms:GenerateDataKey",
                "kms:GenerateDataKeyWithoutPlaintext"
            ],
            "Resource": "arn:aws:kms:us-east-1:444455556666:key/KMS_KEY_ID"
```

```
        }
    ]
}
```

Trust relationship for the Amazon Redshift role

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
                "Service": "redshift.amazonaws.com"
            },
            "Action": "sts:AssumeRole"
        }
    ]
}
```

DynamoDB Table policy (if using resource-based policies)

The following resource-based policy allows a Amazon Redshift service in account 111122223333 to access DynamoDB tables and Streams in account 444455556666. Attach this policy to your DynamoDB table to enable cross-account access.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "AllowRedshiftAccess",
            "Effect": "Allow",
            "Principal": {
                "AWS": "arn:aws:iam::111122223333:role/REDSHIFT_ROLE_NAME"
            },
            "Action": [
```

```
                "dynamodb:DescribeTable",
                "dynamodb:BatchGetItem",
                "dynamodb:Scan",
                "dynamodb:Query",
                "dynamodb:BatchGetItem",
                "dynamodb:GetItem",
                "dynamodb:GetRecords",
                "dynamodb:GetShardIterator",
                "dynamodb:DescribeStream",
                "dynamodb:ListStreams"
            ],
            "Resource": [
                "arn:aws:dynamodb:*:444455556666:table/TABLE_NAME",
                "arn:aws:dynamodb:*:444455556666:table/TABLE_NAME/stream/*"
            ]
        }
    ]
}
```

Important considerations

1. Ensure the KMS key is in the same region as your DynamoDB table.
2. The KMS key must be a customer managed key (CMK), not an AWS managed key.
3. If you're using DynamoDB global tables, configure permissions for all relevant regions.
4. Consider adding condition statements to restrict access based on VPC endpoints or IP ranges.
5. For enhanced security, consider using `aws:PrincipalOrgID` condition to restrict access to your organization.
6. Monitor KMS key usage through CloudTrail and CloudWatch metrics.

DynamoDB zero-ETL integration with Amazon Redshift

Amazon DynamoDB zero-ETL integration with Amazon Redshift enables seamless analytics on DynamoDB data without any coding. This fully-managed feature automatically replicates DynamoDB tables into an Amazon Redshift database so users can run SQL queries and analytics on their DynamoDB data without having to set up complex ETL processes. The integration works by replicating data from the DynamoDB table to the Amazon Redshift database.

To set up the integration, simply specify a DynamoDB table as the source and an Amazon Redshift database as the target. On activation, the integration exports the full DynamoDB table to populate the Amazon Redshift database. The time it takes for this initial process to complete depends on the DynamoDB table size. The zero-ETL integration then incrementally replicates updates from DynamoDB to Amazon Redshift every 15-30 minutes using DynamoDB incremental exports. This means the replicated DynamoDB data in Amazon Redshift is kept up-to-date automatically.

Once configured, users can analyze the DynamoDB data in Amazon Redshift using standard SQL clients and tools, without impacting DynamoDB table performance. By eliminating cumbersome ETL, this zero-ETL integration provides a fast, easy way to unlock insights from DynamoDB through Amazon Redshift analytics and machine learning capabilities.

Topics

- Prerequisites before creating a DynamoDB zero-ETL integration with Amazon Redshift
- Limitations when using DynamoDB zero-ETL integrations with Amazon Redshift
- Creating a DynamoDB zero-ETL integration with Amazon Redshift
- Viewing DynamoDB zero-ETL integrations with Amazon Redshift
- Deleting DynamoDB zero-ETL integrations with Amazon Redshift

Prerequisites before creating a DynamoDB zero-ETL integration with Amazon Redshift

1. You must have your source DynamoDB table and target Amazon Redshift cluster created before creating an integration. This information is covered in Step 1: Configuring a source DynamoDB table and Step 2: Creating an Amazon Redshift data warehouse.
2. A zero-ETL integration between Amazon DynamoDB and Amazon Redshift requires your source DynamoDB table to have Point-in-time recovery (PITR) enabled.
3. For **resource-based policies**, the zero-ETL integration requires a resource-based policy attached directly to your DynamoDB table. This inline policy grants the Amazon Redshift service permission to access your table data for replication. For more information about resource-based policies for DynamoDB, see Using resource-based policies for DynamoDB.

   If you create the integration where your DynamoDB table and Amazon Redshift data warehouse are in the same account, you can use the **Fix it for me** option during the create integration step to automatically apply the required resource policies to both DynamoDB and Amazon Redshift.

If you create an integration where your DynamoDB table and Amazon Redshift data warehouse are in different AWS accounts, you will need to manually apply the following resource policy on your DynamoDB table.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid":
 "StatementthatallowsAmazonRedshiftservicetoDescribeTableandExportTable",
            "Effect": "Allow",
            "Principal": {
                "Service": "redshift.amazonaws.com"
            },
            "Action": [
                "dynamodb:ExportTableToPointInTime",
                "dynamodb:DescribeTable"
            ],
            "Resource": "*",
            "Condition": {
                "StringEquals": {
                    "aws:SourceAccount": "111122223333"
                },
                "ArnEquals": {
                    "aws:SourceArn": "arn:aws:redshift:us-
east-1:111122223333:integration:*"
                }
            }
        },
        {
            "Sid":
 "StatementthatallowsAmazonRedshiftservicetoDescribeTableandExportTable",
            "Effect": "Allow",
            "Principal": {
                "Service": "redshift.amazonaws.com"
            },
            "Action": "dynamodb:DescribeExport",
            "Resource": "arn:aws:dynamodb:us-east-1:111122223333:table/table-
name/export/*",
            "Condition": {
                "StringEquals": {
```

```
                    "aws:SourceAccount": "111122223333"
                },
                "ArnEquals": {
                    "aws:SourceArn": "arn:aws:redshift:us-
east-1:111122223333:integration:*"
                }
            }
        }
    ]
}
```

You may also need to configure the resource policy on your Amazon Redshift data warehouse. For more information, see Configure authorization using the Amazon Redshift API.

4. **For Identity-based policies:**

a. The user creating the integration requires an identity-based policy that authorizes the following actions: `GetResourcePolicy`, `PutResourcePolicy`, and `UpdateContinuousBackups`.

> **Note**
>
> The following policy examples will show the resource as `arn:aws:redshift{-serverless}`. This is an example to show that the arn can be either `arn:aws:redshift` or `arn:aws:redshift-serverless` depending on if your namespace is an Amazon Redshift cluster or Amazon Redshift Serverless namespace.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "dynamodb:ListTables"
            ],
            "Resource": "*"
        },
```

```
        {
            "Effect": "Allow",
            "Action": [
                "dynamodb:GetResourcePolicy",
                "dynamodb:PutResourcePolicy",
                "dynamodb:UpdateContinuousBackups"
            ],
            "Resource": [
                "arn:aws:dynamodb:us-east-1:111122223333:table/table-name"
            ]
        },
        {
            "Sid": "AllowRedshiftDescribeIntegration",
            "Effect": "Allow",
            "Action": [
                "redshift:DescribeIntegrations"
            ],
            "Resource": "*"
        },
        {
            "Sid": "AllowRedshiftCreateIntegration",
            "Effect": "Allow",
            "Action": "redshift:CreateIntegration",
            "Resource": "arn:aws:redshift:us-
east-1:111122223333:integration:*"
        },
        {
            "Sid": "AllowRedshiftModifyDeleteIntegration",
            "Effect": "Allow",
            "Action": [
                "redshift:ModifyIntegration",
                "redshift:DeleteIntegration"
            ],
            "Resource": "arn:aws:redshift:us-
east-1:111122223333:integration:uuid"
        },
        {
            "Sid": "AllowRedshiftCreateInboundIntegration",
            "Effect": "Allow",
            "Action": "redshift:CreateInboundIntegration",
            "Resource": "arn:aws:redshift:us-
east-1:111122223333:namespace:uuid"
        }
    ]
```

```
}
```

b. The user responsible for configuring the destination Amazon Redshift namespace requires an identity-based policy that authorizes the following actions: `PutResourcePolicy`, `DeleteResourcePolicy`, and `GetResourcePolicy`.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "redshift:PutResourcePolicy",
                "redshift:DeleteResourcePolicy",
                "redshift:GetResourcePolicy"
            ],
            "Resource": [
                "arn:aws:redshift:us-east-1:111122223333:cluster:cluster-
name"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "redshift:DescribeInboundIntegrations"
            ],
            "Resource": [
                "arn:aws:redshift:us-east-1:111122223333:cluster:cluster-
name"
            ]
        }
    ]
}
```

5. **Encryption key permissions**

If the source DynamoDB table is encrypted using customer managed AWS KMS key, you will need to add the following policy on your KMS key. This policy allows Amazon Redshift to be able to export data from your encrypted table using your KMS key.

```
{
```

```
    "Sid": "Statement to allow Amazon Redshift service to perform Decrypt operation
 on the source DynamoDB Table",
    "Effect": "Allow",
    "Principal": {
        "Service": [
            "redshift.amazonaws.com"
        ]
    },
    "Action": "kms:Decrypt",
    "Resource": "*",
    "Condition": {
        "StringEquals": {
            "aws:SourceAccount": "<account>"
        },
        "ArnEquals": {
            "aws:SourceArn": "arn:aws:redshift:<region>:<account>:integration:*"
        }
    }
}
```

You can also follow the steps on Getting started with zero-ETL integrations in the Amazon Redshift management guide to configure the permissions of the Amazon Redshift namespace.

Limitations when using DynamoDB zero-ETL integrations with Amazon Redshift

The following general limitations apply to the current release of this integration. These limitations can change in subsequent releases.

> **Note**
>
> In addition to the limitations below, also review the general considerations when using zero-ETL integrations see Considerations when using zero-ETL integrations with Amazon Redshift in the *Amazon Redshift Management Guide*.

- The DynamoDB table and Amazon Redshift cluster need to be in the same Region.
- The source DynamoDB table must be encrypted with either an Amazon-owned or Customer-managed AWS KMS key. Amazon managed encryption is not supported for the source DynamoDB table.

Creating a DynamoDB zero-ETL integration with Amazon Redshift

Before creating a zero-ETL integration, you must first set up your source DynamoDB table and then the target Amazon Redshift data warehouse.

Step 1: Configuring a source DynamoDB table

To create a zero-ETL integration with Amazon Redshift, you need to enable point-in-time recovery (PITR) on your table. If you do not have PITR turned on, the console can fix this for you during the integration setup process. For details on how to enable PITR, see Point-in-time recovery.

Step 2: Creating an Amazon Redshift data warehouse

If you don't already have an Amazon Redshift data warehouse, you can create one. To create an Amazon Redshift Serverless workgroup, see Creating a workgroup with a namespace. To create an Amazon Redshift cluster, see Creating a cluster.

The target Amazon Redshift workgroup or cluster must have the enable_case_sensitive_identifier parameter turned on for the integration to be successful. For more information on enabling case sensitivity, see Turn on case sensitivity for your data warehouse in the Amazon Redshift management guide.

After the Amazon Redshift workgroup or cluster setup is complete, you need to configure your data warehouse. See Zero-ETL integrations in the Amazon Redshift Management Guide for more information.

Step 3: Creating a DynamoDB zero-ETL integration

Before you create a zero-ETL integration, make sure to complete the tasks in the section titled Prerequisites before creating a DynamoDB zero-ETL integration with Amazon Redshift. Creating an integration between DynamoDB and Amazon Redshift is a two-step process. First create an integration from the DynamoDB, and then attach a Amazon Redshift database to this newly created integration.

Create a zero-ETL integration

1. Sign in to the AWS Management Console and open the Amazon DynamoDB console at https://console.aws.amazon.com/dynamodbv2.
2. In the navigation pane, choose **Integrations**.

3. Select **Create zero-ETL integration** and choose **Amazon Redshift**.
4. This will take you to the **Amazon Redshift console**. To continue with the procedure, see the **DynamoDB section** in Create a zero-ETL integration for DynamoDB.

Viewing DynamoDB zero-ETL integrations with Amazon Redshift

You can view the details of a zero-ETL integration to see its configuration information and current status.

To view the details of a zero-ETL integration in the Amazon DynamoDB console:

1. Sign in to the AWS Management Console and open the Amazon DynamoDB console at https://console.aws.amazon.com/dynamodbv2.
2. In the DynamoDB console, choose **Integrations**.
3. In the **Zero-ETL integration** pane, select the zero-ETL integration you want to view.

To view the details of a zero-ETL integration in the Amazon Redshift console:

1. Sign in to the AWS Management Console and open the Amazon Redshift console at https://console.aws.amazon.com/redshiftv2.
2. Follow the steps at Viewing zero-ETL integrations.

> **Note**
>
> The possible statuses of a zero-ETL integration with Amazon Redshift are listed in Viewing zero-ETL integrations in the *Amazon Redshift Management Guide*.

Deleting DynamoDB zero-ETL integrations with Amazon Redshift

When you delete a zero-ETL integration, your data isn't deleted from DynamoDB or Amazon Redshift, but DynamoDB stops sending data from your source table to the Amazon Redshift target.

To delete a zero-ETL integration

1. Sign in to the AWS Management Console and open the Amazon DynamoDB console at https://console.aws.amazon.com/dynamodbv2.

2. In the DynamoDB console, choose **Integrations**.
3. In the **Zero-ETL integration** pane, select the zero-ETL integration you want to delete.
4. Choose **Manage**. This will take you to the integration details page.
5. To confirm the deletion, choose **Delete**.

Loading data from DynamoDB into Amazon Redshift with the COPY command

Amazon Redshift works with Amazon DynamoDB with advanced business intelligence capabilities and a powerful SQL-based interface. When you copy data from a DynamoDB table into Amazon Redshift, you can perform complex data analysis queries on that data, including joins with other tables in your Amazon Redshift cluster.

In terms of provisioned throughput, a copy operation from a DynamoDB table counts against that table's read capacity. After the data is copied, your SQL queries in Amazon Redshift do not affect DynamoDB in any way. This is because your queries act upon a copy of the data from DynamoDB, rather than upon DynamoDB itself.

Before you can load data from a DynamoDB table, you must first create an Amazon Redshift table to serve as the destination for the data. Keep in mind that you are copying data from a NoSQL environment into a SQL environment, and that there are certain rules in one environment that do not apply in the other. Here are some of the differences to consider:

- DynamoDB table names can contain up to 255 characters, including '.' (dot) and '-' (dash) characters, and are case-sensitive. Amazon Redshift table names are limited to 127 characters, cannot contain dots or dashes and are not case-sensitive. In addition, table names cannot conflict with any Amazon Redshift reserved words.
- DynamoDB does not support the SQL concept of NULL. You need to specify how Amazon Redshift interprets empty or blank attribute values in DynamoDB, treating them either as NULLs or as empty fields.
- DynamoDB data types do not correspond directly with those of Amazon Redshift. You need to ensure that each column in the Amazon Redshift table is of the correct data type and size to accommodate the data from DynamoDB.

Here is an example COPY command from Amazon Redshift SQL:

```
copy favoritemovies from 'dynamodb://my-favorite-movies-table'
credentials 'aws_access_key_id=<Your-Access-Key-ID>;aws_secret_access_key=<Your-Secret-
Access-Key>'
readratio 50;
```

In this example, the source table in DynamoDB is `my-favorite-movies-table`. The target table in Amazon Redshift is `favoritemovies`. The `readratio 50` clause regulates the percentage of provisioned throughput that is consumed; in this case, the COPY command will use no more than 50 percent of the read capacity units provisioned for `my-favorite-movies-table`. We highly recommend setting this ratio to a value less than the average unused provisioned throughput.

For detailed instructions on loading data from DynamoDB into Amazon Redshift, refer to the following sections in the *Amazon Redshift Database Developer Guide*:

- Loading data from a DynamoDB table
- The COPY command
- COPY examples

Processing DynamoDB data with Apache Hive on Amazon EMR

Amazon DynamoDB is integrated with Apache Hive, a data warehousing application that runs on Amazon EMR. Hive can read and write data in DynamoDB tables, allowing you to:

- Query live DynamoDB data using a SQL-like language (HiveQL).
- Copy data from a DynamoDB table to an Amazon S3 bucket, and vice-versa.
- Copy data from a DynamoDB table into Hadoop Distributed File System (HDFS), and vice-versa.
- Perform join operations on DynamoDB tables.

Topics

- Overview
- Tutorial: Working with Amazon DynamoDB and Apache Hive
- Creating an external table in Hive
- Processing HiveQL statements
- Querying data in DynamoDB

- Copying data to and from Amazon DynamoDB
- Performance tuning

Overview

Amazon EMR is a service that makes it easy to quickly and cost-effectively process vast amounts of data. To use Amazon EMR, you launch a managed cluster of Amazon EC2 instances running the Hadoop open source framework. *Hadoop* is a distributed application that implements the MapReduce algorithm, where a task is mapped to multiple nodes in the cluster. Each node processes its designated work, in parallel with the other nodes. Finally, the outputs are reduced on a single node, yielding the final result.

You can choose to launch your Amazon EMR cluster so that it is persistent or transient:

- A *persistent* cluster runs until you shut it down. Persistent clusters are ideal for data analysis, data warehousing, or any other interactive use.
- A *transient* cluster runs long enough to process a job flow, and then shuts down automatically. Transient clusters are ideal for periodic processing tasks, such as running scripts.

For information about Amazon EMR architecture and administration, see the Amazon EMR Management Guide.

When you launch an Amazon EMR cluster, you specify the initial number and type of Amazon EC2 instances. You also specify other distributed applications (in addition to Hadoop itself) that you want to run on the cluster. These applications include Hue, Mahout, Pig, Spark, and more.

For information about applications for Amazon EMR, see the Amazon EMR Release Guide.

Depending on the cluster configuration, you might have one or more of the following node types:

- Leader node — Manages the cluster, coordinating the distribution of the MapReduce executable and subsets of the raw data, to the core and task instance groups. It also tracks the status of each task performed and monitors the health of the instance groups. There is only one leader node in a cluster.
- Core nodes — Runs MapReduce tasks and stores data using the Hadoop Distributed File System (HDFS).
- Task nodes (optional) — Runs MapReduce tasks.

Tutorial: Working with Amazon DynamoDB and Apache Hive

In this tutorial, you will launch an Amazon EMR cluster, and then use Apache Hive to process data stored in a DynamoDB table.

Hive is a data warehouse application for Hadoop that allows you to process and analyze data from multiple sources. Hive provides a SQL-like language, *HiveQL*, that lets you work with data stored locally in the Amazon EMR cluster or in an external data source (such as Amazon DynamoDB).

For more information, see to the Hive Tutorial.

Topics

- Before you begin
- Step 1: Create an Amazon EC2 key pair
- Step 2: Launch an Amazon EMR cluster
- Step 3: Connect to the Leader node
- Step 4: Load data into HDFS
- Step 5: Copy data to DynamoDB
- Step 6: Query the data in the DynamoDB table
- Step 7: (Optional) clean up

Before you begin

For this tutorial, you will need the following:

- An AWS account. If you do not have one, see Signing up for AWS.
- An SSH client (Secure Shell). You use the SSH client to connect to the leader node of the Amazon EMR cluster and run interactive commands. SSH clients are available by default on most Linux, Unix, and Mac OS X installations. Windows users can download and install the PuTTY client, which has SSH support.

Next step

Step 1: Create an Amazon EC2 key pair

Step 1: Create an Amazon EC2 key pair

In this step, you will create the Amazon EC2 key pair you need to connect to an Amazon EMR leader node and run Hive commands.

1. Sign in to the AWS Management Console and open the Amazon EC2 console at https://console.aws.amazon.com/ec2/.
2. Choose a region (for example, `US West (Oregon)`). This should be the same region in which your DynamoDB table is located.
3. In the navigation pane, choose **Key Pairs**.
4. Choose **Create Key Pair**.
5. In **Key pair name**, type a name for your key pair (for example, `mykeypair`), and then choose **Create**.
6. Download the private key file. The file name will end with `.pem` (such as `mykeypair.pem`). Keep this private key file in a safe place. You will need it to access any Amazon EMR cluster that you launch with this key pair.

   > **Important**
   >
   > If you lose the key pair, you cannot connect to the leader node of your Amazon EMR cluster.

   For more information about key pairs, see Amazon EC2 Key Pairs in the *Amazon EC2 User Guide*.

Next step

Step 2: Launch an Amazon EMR cluster

Step 2: Launch an Amazon EMR cluster

In this step, you will configure and launch an Amazon EMR cluster. Hive and a storage handler for DynamoDB will already be installed on the cluster.

1. Open the Amazon EMR console at https://console.aws.amazon.com/emr.
2. Choose **Create Cluster**.

3. On the **Create Cluster - Quick Options** page, do the following:

   a. In **Cluster name**, type a name for your cluster (for example: `My EMR cluster`).

   b. In **EC2 key pair**, choose the key pair you created earlier.

   Leave the other settings at their defaults.

4. Choose **Create cluster**.

It will take several minutes to launch your cluster. You can use the **Cluster Details** page in the Amazon EMR console to monitor its progress.

When the status changes to `Waiting`, the cluster is ready for use.

Cluster log files and Amazon S3

An Amazon EMR cluster generates log files that contain information about the cluster status and debugging information. The default settings for **Create Cluster - Quick Options** include setting up Amazon EMR logging.

If one does not already exist, the AWS Management Console creates an Amazon S3 bucket. The bucket name is `aws-logs-`*`account-id`*`-`*`region`*, where *`account-id`* is your AWS account number and *`region`* is the region in which you launched the cluster (for example, `aws-logs-123456789012-us-west-2`).

> **Note**
>
> You can use the Amazon S3 console to view the log files. For more information, see View Log Files in the *Amazon EMR Management Guide*.

You can use this bucket for purposes in addition to logging. For example, you can use the bucket as a location for storing a Hive script or as a destination when exporting data from Amazon DynamoDB to Amazon S3.

Next step

Step 3: Connect to the Leader node

Step 3: Connect to the Leader node

When the status of your Amazon EMR cluster changes to `Waiting`, you will be able to connect to the leader node using SSH and perform command line operations.

1. In the Amazon EMR console, choose your cluster's name to view its status.
2. On the **Cluster Details** page, find the **Leader public DNS** field. This is the public DNS name for the leader node of your Amazon EMR cluster.
3. To the right of the DNS name, choose the **SSH** link.
4. Follow the instructions in **Connect to the Leader Node Using SSH** .

   Depending on your operating system, choose the **Windows** tab or the **Mac/Linux** tab, and follow the instructions for connecting to the leader node.

After you connect to the leader node using either SSH or PuTTY, you should see a command prompt similar to the following:

```
[hadoop@ip-192-0-2-0 ~]$
```

Next step

Step 4: Load data into HDFS

Step 4: Load data into HDFS

In this step, you will copy a data file into Hadoop Distributed File System (HDFS), and then create an external Hive table that maps to the data file.

Download the sample data

1. Download the sample data archive (`features.zip`):

   ```
   wget https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/samples/
   features.zip
   ```

2. Extract the `features.txt` file from the archive:

   ```
   unzip features.zip
   ```

3. View the first few lines of the `features.txt` file:

```
head features.txt
```

The result should look similar to this:

```
1535908|Big Run|Stream|WV|38.6370428|-80.8595469|794
875609|Constable Hook|Cape|NJ|40.657881|-74.0990309|7
1217998|Gooseberry Island|Island|RI|41.4534361|-71.3253284|10
26603|Boone Moore Spring|Spring|AZ|34.0895692|-111.410065|3681
1506738|Missouri Flat|Flat|WA|46.7634987|-117.0346113|2605
1181348|Minnow Run|Stream|PA|40.0820178|-79.3800349|1558
1288759|Hunting Creek|Stream|TN|36.343969|-83.8029682|1024
533060|Big Charles Bayou|Bay|LA|29.6046517|-91.9828654|0
829689|Greenwood Creek|Stream|NE|41.596086|-103.0499296|3671
541692|Button Willow Island|Island|LA|31.9579389|-93.0648847|98
```

The `features.txt` file contains a subset of data from the United States Board on Geographic Names (http://geonames.usgs.gov/domestic/download_data.htm). The fields in each line represent the following:

- Feature ID (unique identifier)
- Name
- Class (lake; forest; stream; and so on)
- State
- Latitude (degrees)
- Longitude (degrees)
- Height (in feet)

4. At the command prompt, enter the following command:

```
hive
```

The command prompt changes to this: `hive>`

5. Enter the following HiveQL statement to create a native Hive table:

```
CREATE TABLE hive_features
    (feature_id             BIGINT,
```

```
    feature_name            STRING ,
    feature_class           STRING ,
    state_alpha             STRING,
    prim_lat_dec            DOUBLE ,
    prim_long_dec           DOUBLE ,
    elev_in_ft              BIGINT)
    ROW FORMAT DELIMITED
    FIELDS TERMINATED BY '|'
    LINES TERMINATED BY '\n';
```

6. Enter the following HiveQL statement to load the table with data:

```
LOAD DATA
LOCAL
INPATH './features.txt'
OVERWRITE
INTO TABLE hive_features;
```

7. You now have a native Hive table populated with data from the `features.txt` file. To verify, enter the following HiveQL statement:

```
SELECT state_alpha, COUNT(*)
FROM hive_features
GROUP BY state_alpha;
```

   The output should show a list of states and the number of geographic features in each.

Next step

Step 5: Copy data to DynamoDB

Step 5: Copy data to DynamoDB

In this step, you will copy data from the Hive table (`hive_features`) to a new table in DynamoDB.

1. Open the DynamoDB console at https://console.aws.amazon.com/dynamodb/.
2. Choose **Create Table**.
3. On the **Create DynamoDB table** page, do the following:
   a. In **Table**, type **`Features`**.
   b. For **Primary key**, in the **Partition key** field, type **`Id`**. Set the data type to **Number**.

Clear **Use Default Settings**. For **Provisioned Capacity**, type the following:

- **Read Capacity Units**—10
- **Write Capacity Units**—10

Choose **Create**.

4. At the Hive prompt, enter the following HiveQL statement:

```
CREATE EXTERNAL TABLE ddb_features
    (feature_id   BIGINT,
    feature_name  STRING,
    feature_class STRING,
    state_alpha   STRING,
    prim_lat_dec  DOUBLE,
    prim_long_dec DOUBLE,
    elev_in_ft    BIGINT)
STORED BY 'org.apache.hadoop.hive.dynamodb.DynamoDBStorageHandler'
TBLPROPERTIES(
    "dynamodb.table.name" = "Features",

 "dynamodb.column.mapping"="feature_id:Id,feature_name:Name,feature_class:Class,state_alpha
);
```

You have now established a mapping between Hive and the Features table in DynamoDB.

5. Enter the following HiveQL statement to import data to DynamoDB:

```
INSERT OVERWRITE TABLE ddb_features
SELECT
    feature_id,
    feature_name,
    feature_class,
    state_alpha,
    prim_lat_dec,
    prim_long_dec,
    elev_in_ft
FROM hive_features;
```

Hive will submit a MapReduce job, which will be processed by your Amazon EMR cluster. It will take several minutes to complete the job.

6. Verify that the data has been loaded into DynamoDB:

   a. In the DynamoDB console navigation pane, choose **Tables**.

   b. Choose the Features table, and then choose the **Items** tab to view the data.

Next step

Step 6: Query the data in the DynamoDB table

Step 6: Query the data in the DynamoDB table

In this step, you will use HiveQL to query the Features table in DynamoDB. Try the following Hive queries:

1. All of the feature types (`feature_class`) in alphabetical order:

```
SELECT DISTINCT feature_class
FROM ddb_features
ORDER BY feature_class;
```

2. All of the lakes that begin with the letter "M":

```
SELECT feature_name, state_alpha
FROM ddb_features
WHERE feature_class = 'Lake'
AND feature_name LIKE 'M%'
ORDER BY feature_name;
```

3. States with at least three features higher than a mile (5,280 feet):

```
SELECT state_alpha, feature_class, COUNT(*)
FROM ddb_features
WHERE elev_in_ft > 5280
GROUP by state_alpha, feature_class
HAVING COUNT(*) >= 3
ORDER BY state_alpha, feature_class;
```

Next step

Step 7: (Optional) clean up

Step 7: (Optional) clean up

Now that you have completed the tutorial, you can continue reading this section to learn more about working with DynamoDB data in Amazon EMR. You might decide to keep your Amazon EMR cluster up and running while you do this.

If you don't need the cluster anymore, you should terminate it and remove any associated resources. This will help you avoid being charged for resources you don't need.

1. Terminate the Amazon EMR cluster:
   a. Open the Amazon EMR console at https://console.aws.amazon.com/emr.
   b. Choose the Amazon EMR cluster, choose **Terminate**, and then confirm.
2. Delete the Features table in DynamoDB:
   a. Open the DynamoDB console at https://console.aws.amazon.com/dynamodb/.
   b. In the navigation pane, choose **Tables**.
   c. Choose the Features table. From the **Actions** menu, choose **Delete Table**.
3. Delete the Amazon S3 bucket containing the Amazon EMR log files:
   a. Open the Amazon S3 console at https://console.aws.amazon.com/s3/.
   b. From the list of buckets, choose `aws-logs-` *`accountID`*`-`*`region`*, where *`accountID`* is your AWS account number and *`region`* is the region in which you launched the cluster.
   c. From the **Action** menu, choose **Delete**.

Creating an external table in Hive

In Tutorial: Working with Amazon DynamoDB and Apache Hive, you created an external Hive table that mapped to a DynamoDB table. When you issued HiveQL statements against the external table, the read and write operations were passed through to the DynamoDB table.

You can think of an external table as a pointer to a data source that is managed and stored elsewhere. In this case, the underlying data source is a DynamoDB table. (The table must already exist. You cannot create, update, or delete a DynamoDB table from within Hive.) You use the `CREATE EXTERNAL TABLE` statement to create the external table. After that, you can use HiveQL to work with data in DynamoDB, as if that data were stored locally within Hive.

> **Note**
>
> You can use INSERT statements to insert data into an external table and SELECT statements to select data from it. However, you cannot use UPDATE or DELETE statements to manipulate data in the table.

If you no longer need the external table, you can remove it using the DROP TABLE statement. In this case, DROP TABLE only removes the external table in Hive. It does not affect the underlying DynamoDB table or any of its data.

Topics

- CREATE EXTERNAL TABLE syntax
- Data type mappings

CREATE EXTERNAL TABLE syntax

The following shows the HiveQL syntax for creating an external Hive table that maps to a DynamoDB table:

```
CREATE EXTERNAL TABLE hive_table

 (hive_column1_name hive_column1_datatype, hive_column2_name hive_column2_datatype...)
STORED BY 'org.apache.hadoop.hive.dynamodb.DynamoDBStorageHandler'
TBLPROPERTIES (
    "dynamodb.table.name" = "dynamodb_table",
    "dynamodb.column.mapping" =
 "hive_column1_name:dynamodb_attribute1_name,hive_column2_name:dynamodb_attribute2_name..."
);
```

Line 1 is the start of the CREATE EXTERNAL TABLE statement, where you provide the name of the Hive table (*hive_table*) you want to create.

Line 2 specifies the columns and data types for *hive_table*. You need to define columns and data types that correspond to the attributes in the DynamoDB table.

Line 3 is the STORED BY clause, where you specify a class that handles data management between the Hive and the DynamoDB table. For DynamoDB, STORED BY should be set to `'org.apache.hadoop.hive.dynamodb.DynamoDBStorageHandler'`.

Line 4 is the start of the `TBLPROPERTIES` clause, where you define the following parameters for `DynamoDBStorageHandler`:

- `dynamodb.table.name`—the name of the DynamoDB table.
- `dynamodb.column.mapping`—pairs of column names in the Hive table and their corresponding attributes in the DynamoDB table. Each pair is of the form *hive_column_name:dynamodb_attribute_name*, and the pairs are separated by commas.

Note the following:

- The name of the Hive table name does not have to be the same as the DynamoDB table name.
- The Hive table column names do not have to be the same as those in the DynamoDB table.
- The table specified by `dynamodb.table.name` must exist in DynamoDB.
- For `dynamodb.column.mapping`:
  - You must map the key schema attributes for the DynamoDB table. This includes the partition key and the sort key (if present).
  - You do not have to map the non-key attributes of the DynamoDB table. However, you will not see any data from those attributes when you query the Hive table.
  - If the data types of a Hive table column and a DynamoDB attribute are incompatible, you will see `NULL` in these columns when you query the Hive table.

> **Note**
>
> The `CREATE EXTERNAL TABLE` statement does not perform any validation on the `TBLPROPERTIES` clause. The values you provide for `dynamodb.table.name` and `dynamodb.column.mapping` are only evaluated by the `DynamoDBStorageHandler` class when you attempt to access the table.

Data type mappings

The following table shows DynamoDB data types and compatible Hive data types:

DynamoDB Data Type	Hive Data Type
String	`STRING`
Number	`BIGINT` or `DOUBLE`
Binary	`BINARY`
String Set	`ARRAY<STRING>`
Number Set	`ARRAY<BIGINT>` or `ARRAY<DOUBLE>`
Binary Set	`ARRAY<BINARY>`

Note

The following DynamoDB data types are not supported by the `DynamoDBStorageHandler` class, so they cannot be used with `dynamodb.column.mapping`:

- Map
- List
- Boolean
- Null

However, if you need to work with these data types, you can create a single entity called `item` that represents the entire DynamoDB item as a map of strings for both keys and values in the map. For more information, see Copying data without a column mapping

If you want to map a DynamoDB attribute of type Number, you must choose an appropriate Hive type:

- The Hive `BIGINT` type is for 8-byte signed integers. It is the same as the `long` data type in Java.
- The Hive `DOUBLE` type is for 8-bit double precision floating point numbers. It is the same as the `double` type in Java.

If you have numeric data stored in DynamoDB that has a higher precision than the Hive data type you choose, then accessing the DynamoDB data could cause a loss of precision.

If you export data of type Binary from DynamoDB to (Amazon S3) or HDFS, the data is stored as a Base64-encoded string. If you import data from Amazon S3 or HDFS into the DynamoDB Binary type, you must ensure the data is encoded as a Base64 string.

Processing HiveQL statements

Hive is an application that runs on Hadoop, which is a batch-oriented framework for running MapReduce jobs. When you issue a HiveQL statement, Hive determines whether it can return the results immediately or whether it must submit a MapReduce job.

For example, consider the *ddb_features* table (from Tutorial: Working with Amazon DynamoDB and Apache Hive). The following Hive query prints state abbreviations and the number of summits in each:

```
SELECT state_alpha, count(*)
FROM ddb_features
WHERE feature_class = 'Summit'
GROUP BY state_alpha;
```

Hive does not return the results immediately. Instead, it submits a MapReduce job, which is processed by the Hadoop framework. Hive will wait until the job is complete before it shows the results from the query:

```
AK  2
AL  2
AR  2
AZ  3
CA  7
CO  2
CT  2
ID  1
KS  1
ME  2
MI  1
MT  3
NC  1
NE  1
NM  1
```

```
NY  2
OR  5
PA  1
TN  1
TX  1
UT  4
VA  1
VT  2
WA  2
WY  3
Time taken: 8.753 seconds, Fetched: 25 row(s)
```

Monitoring and canceling jobs

When Hive launches a Hadoop job, it prints output from that job. The job completion status is updated as the job progresses. In some cases, the status might not be updated for a long time. (This can happen when you are querying a large DynamoDB table that has a low provisioned read capacity setting.)

If you need to cancel the job before it is complete, you can type **`Ctrl+C`** at any time.

Querying data in DynamoDB

The following examples show some ways that you can use HiveQL to query data stored in DynamoDB.

These examples refer to the *ddb_features* table in the tutorial (Step 5: Copy data to DynamoDB).

Topics

- Using aggregate functions
- Using the GROUP BY and HAVING clauses
- Joining two DynamoDB tables
- Joining tables from different sources

Using aggregate functions

HiveQL provides built-in functions for summarizing data values. For example, you can use the `MAX` function to find the largest value for a selected column. The following example returns the elevation of the highest feature in the state of Colorado.

```
SELECT MAX(elev_in_ft)
FROM ddb_features
WHERE state_alpha = 'CO';
```

Using the GROUP BY and HAVING clauses

You can use the `GROUP BY` clause to collect data across multiple records. This is often used with an aggregate function such as `SUM`, `COUNT`, `MIN`, or `MAX`. You can also use the `HAVING` clause to discard any results that do not meet certain criteria.

The following example returns a list of the highest elevations from states that have more than five features in the *ddb_features* table.

```
SELECT state_alpha, max(elev_in_ft)
FROM ddb_features
GROUP BY state_alpha
HAVING count(*) >= 5;
```

Joining two DynamoDB tables

The following example maps another Hive table (*east_coast_states*) to a table in DynamoDB. The `SELECT` statement is a join across these two tables. The join is computed on the cluster and returned. The join does not take place in DynamoDB.

Consider a DynamoDB table named EastCoastStates that contains the following data:

```
StateName       StateAbbrev

Maine           ME
New Hampshire   NH
Massachusetts   MA
Rhode Island    RI
Connecticut     CT
New York        NY
New Jersey      NJ
Delaware        DE
Maryland        MD
Virginia        VA
North Carolina  NC
South Carolina  SC
Georgia         GA
```

```
Florida         FL
```

Let's assume the table is available as a Hive external table named east_coast_states:

```
CREATE EXTERNAL TABLE ddb_east_coast_states (state_name STRING, state_alpha STRING)
STORED BY 'org.apache.hadoop.hive.dynamodb.DynamoDBStorageHandler'
TBLPROPERTIES ("dynamodb.table.name" = "EastCoastStates",
"dynamodb.column.mapping" = "state_name:StateName,state_alpha:StateAbbrev");
```

The following join returns the states on the East Coast of the United States that have at least three features:

```
SELECT ecs.state_name, f.feature_class, COUNT(*)
FROM ddb_east_coast_states ecs
JOIN ddb_features f on ecs.state_alpha = f.state_alpha
GROUP BY ecs.state_name, f.feature_class
HAVING COUNT(*) >= 3;
```

Joining tables from different sources

In the following example, s3_east_coast_states is a Hive table associated with a CSV file stored in Amazon S3. The *ddb_features* table is associated with data in DynamoDB. The following example joins these two tables, returning the geographic features from states whose names begin with "New."

```
create external table s3_east_coast_states (state_name STRING, state_alpha STRING)
ROW FORMAT DELIMITED FIELDS TERMINATED BY ','
LOCATION 's3://bucketname/path/subpath/';
```

```
SELECT ecs.state_name, f.feature_name, f.feature_class
FROM s3_east_coast_states ecs
JOIN ddb_features f
ON ecs.state_alpha = f.state_alpha
WHERE ecs.state_name LIKE 'New%';
```

Copying data to and from Amazon DynamoDB

In the Tutorial: Working with Amazon DynamoDB and Apache Hive, you copied data from a native Hive table into an external DynamoDB table, and then queried the external DynamoDB table. The

table is external because it exists outside of Hive. Even if you drop the Hive table that maps to it, the table in DynamoDB is not affected.

Hive is an excellent solution for copying data among DynamoDB tables, Amazon S3 buckets, native Hive tables, and Hadoop Distributed File System (HDFS). This section provides examples of these operations.

Topics

- Copying data between DynamoDB and a native Hive table
- Copying data between DynamoDB and Amazon S3
- Copying data between DynamoDB and HDFS
- Using data compression
- Reading non-printable UTF-8 character data

Copying data between DynamoDB and a native Hive table

If you have data in a DynamoDB table, you can copy the data to a native Hive table. This will give you a snapshot of the data, as of the time you copied it.

You might decide to do this if you need to perform many HiveQL queries, but do not want to consume provisioned throughput capacity from DynamoDB. Because the data in the native Hive table is a copy of the data from DynamoDB, and not "live" data, your queries should not expect that the data is up-to-date.

> **Note**
>
> The examples in this section are written with the assumption you followed the steps in Tutorial: Working with Amazon DynamoDB and Apache Hive and have an external table in DynamoDB named *ddb_features*.

Example From DynamoDB to native Hive table

You can create a native Hive table and populate it with data from *ddb_features*, like this:

```
CREATE TABLE features_snapshot AS
SELECT * FROM ddb_features;
```

You can then refresh the data at any time:

```
INSERT OVERWRITE TABLE features_snapshot
SELECT * FROM ddb_features;
```

In these examples, the subquery `SELECT * FROM ddb_features` will retrieve all of the data from *ddb_features*. If you only want to copy a subset of the data, you can use a `WHERE` clause in the subquery.

The following example creates a native Hive table, containing only some of the attributes for lakes and summits:

```
CREATE TABLE lakes_and_summits AS
SELECT feature_name, feature_class, state_alpha
FROM ddb_features
WHERE feature_class IN ('Lake','Summit');
```

Example From native Hive table to DynamoDB

Use the following HiveQL statement to copy the data from the native Hive table to *ddb_features*:

```
INSERT OVERWRITE TABLE ddb_features
SELECT * FROM features_snapshot;
```

Copying data between DynamoDB and Amazon S3

If you have data in a DynamoDB table, you can use Hive to copy the data to an Amazon S3 bucket.

You might do this if you want to create an archive of data in your DynamoDB table. For example, suppose you have a test environment where you need to work with a baseline set of test data in DynamoDB. You can copy the baseline data to an Amazon S3 bucket, and then run your tests. Afterward, you can reset the test environment by restoring the baseline data from the Amazon S3 bucket to DynamoDB.

If you worked through Tutorial: Working with Amazon DynamoDB and Apache Hive, then you already have an Amazon S3 bucket that contains your Amazon EMR logs. You can use this bucket for the examples in this section, if you know the root path for the bucket:

1. Open the Amazon EMR console at https://console.aws.amazon.com/emr.

2. For **Name**, choose your cluster.
3. The URI is listed in **Log URI** under **Configuration Details**.
4. Make a note of the root path of the bucket. The naming convention is:

   `s3://aws-logs-`*`accountID`*`-`*`region`*

   where *`accountID`* is your AWS account ID and region is the AWS region for the bucket.

> **Note**
>
> For these examples, we will use a subpath within the bucket, as in this example: `s3://aws-logs-123456789012-us-west-2/hive-test`

The following procedures are written with the assumption you followed the steps in the tutorial and have an external table in DynamoDB named *ddb_features*.

Topics

- Copying data using the Hive default format
- Copying data with a user-specified format
- Copying data without a column mapping
- Viewing the data in Amazon S3

Copying data using the Hive default format

Example From DynamoDB to Amazon S3

Use an `INSERT OVERWRITE` statement to write directly to Amazon S3.

```
INSERT OVERWRITE DIRECTORY 's3://aws-logs-123456789012-us-west-2/hive-test'
SELECT * FROM ddb_features;
```

The data file in Amazon S3 looks like this:

```
920709^ASoldiers Farewell Hill^ASummit^ANM^A32.3564729^A-108.33004616135
1178153^AJones Run^AStream^APA^A41.2120086^A-79.25920781260
253838^ASentinel Dome^ASummit^ACA^A37.7229821^A-119.584338133
```

```
264054^ANeversweet Gulch^AValley^ACA^A41.6565269^A-122.83614322900
115905^AChacaloochee Bay^ABay^AAL^A30.6979676^A-87.97388530
```

Each field is separated by an SOH character (start of heading, 0x01). In the file, SOH appears as **^A**.

Example From Amazon S3 to DynamoDB

1. Create an external table pointing to the unformatted data in Amazon S3.

```
CREATE EXTERNAL TABLE s3_features_unformatted
    (feature_id       BIGINT,
    feature_name      STRING ,
    feature_class     STRING ,
    state_alpha       STRING,
    prim_lat_dec      DOUBLE ,
    prim_long_dec     DOUBLE ,
    elev_in_ft        BIGINT)
LOCATION 's3://aws-logs-123456789012-us-west-2/hive-test';
```

2. Copy the data to DynamoDB.

```
INSERT OVERWRITE TABLE ddb_features
SELECT * FROM s3_features_unformatted;
```

Copying data with a user-specified format

If you want to specify your own field separator character, you can create an external table that maps to the Amazon S3 bucket. You might use this technique for creating data files with comma-separated values (CSV).

Example From DynamoDB to Amazon S3

1. Create a Hive external table that maps to Amazon S3. When you do this, ensure that the data types are consistent with those of the DynamoDB external table.

```
CREATE EXTERNAL TABLE s3_features_csv
    (feature_id       BIGINT,
    feature_name      STRING,
    feature_class     STRING,
    state_alpha       STRING,
    prim_lat_dec      DOUBLE,
```

```
    prim_long_dec     DOUBLE,
    elev_in_ft        BIGINT)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
LOCATION 's3://aws-logs-123456789012-us-west-2/hive-test';
```

2. Copy the data from DynamoDB.

```
INSERT OVERWRITE TABLE s3_features_csv
SELECT * FROM ddb_features;
```

The data file in Amazon S3 looks like this:

```
920709,Soldiers Farewell Hill,Summit,NM,32.3564729,-108.3300461,6135
1178153,Jones Run,Stream,PA,41.2120086,-79.2592078,1260
253838,Sentinel Dome,Summit,CA,37.7229821,-119.58433,8133
264054,Neversweet Gulch,Valley,CA,41.6565269,-122.8361432,2900
115905,Chacaloochee Bay,Bay,AL,30.6979676,-87.9738853,0
```

Example From Amazon S3 to DynamoDB

With a single HiveQL statement, you can populate the DynamoDB table using the data from Amazon S3:

```
INSERT OVERWRITE TABLE ddb_features
SELECT * FROM s3_features_csv;
```

Copying data without a column mapping

You can copy data from DynamoDB in a raw format and write it to Amazon S3 without specifying any data types or column mapping. You can use this method to create an archive of DynamoDB data and store it in Amazon S3.

Example From DynamoDB to Amazon S3

1. Create an external table associated with your DynamoDB table. (There is no `dynamodb.column.mapping` in this HiveQL statement.)

```
CREATE EXTERNAL TABLE ddb_features_no_mapping
```

```
    (item MAP<STRING, STRING>)
STORED BY 'org.apache.hadoop.hive.dynamodb.DynamoDBStorageHandler'
TBLPROPERTIES ("dynamodb.table.name" = "Features");
```

2. Create another external table associated with your Amazon S3 bucket.

```
CREATE EXTERNAL TABLE s3_features_no_mapping
    (item MAP<STRING, STRING>)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\t'
LINES TERMINATED BY '\n'
LOCATION 's3://aws-logs-123456789012-us-west-2/hive-test';
```

3. Copy the data from DynamoDB to Amazon S3.

```
INSERT OVERWRITE TABLE s3_features_no_mapping
SELECT * FROM ddb_features_no_mapping;
```

The data file in Amazon S3 looks like this:

```
Name^C{"s":"Soldiers Farewell
 Hill"}^BState^C{"s":"NM"}^BClass^C{"s":"Summit"}^BElevation^C{"n":"6135"}^BLatitude^C{"n":"32.
Name^C{"s":"Jones
 Run"}^BState^C{"s":"PA"}^BClass^C{"s":"Stream"}^BElevation^C{"n":"1260"}^BLatitude^C{"n":"41.2
Name^C{"s":"Sentinel
 Dome"}^BState^C{"s":"CA"}^BClass^C{"s":"Summit"}^BElevation^C{"n":"8133"}^BLatitude^C{"n":"37.
Name^C{"s":"Neversweet
 Gulch"}^BState^C{"s":"CA"}^BClass^C{"s":"Valley"}^BElevation^C{"n":"2900"}^BLatitude^C{"n":"41
Name^C{"s":"Chacaloochee
 Bay"}^BState^C{"s":"AL"}^BClass^C{"s":"Bay"}^BElevation^C{"n":"0"}^BLatitude^C{"n":"30.6979676
```

Each field begins with an STX character (start of text, 0x02) and ends with an ETX character (end of text, 0x03). In the file, STX appears as **^B** and ETX appears as **^C**.

Example From Amazon S3 to DynamoDB

With a single HiveQL statement, you can populate the DynamoDB table using the data from Amazon S3:

```
INSERT OVERWRITE TABLE ddb_features_no_mapping
```

```
SELECT * FROM s3_features_no_mapping;
```

Viewing the data in Amazon S3

If you use SSH to connect to the leader node, you can use the AWS Command Line Interface (AWS CLI) to access the data that Hive wrote to Amazon S3.

The following steps are written with the assumption you have copied data from DynamoDB to Amazon S3 using one of the procedures in this section.

1. If you are currently at the Hive command prompt, exit to the Linux command prompt.

   ```
   hive> exit;
   ```

2. List the contents of the hive-test directory in your Amazon S3 bucket. (This is where Hive copied the data from DynamoDB.)

   ```
   aws s3 ls s3://aws-logs-123456789012-us-west-2/hive-test/
   ```

   The response should look similar to this:

   ```
   2016-11-01 23:19:54 81983 000000_0
   ```

   The file name (*000000_0*) is system-generated.

3. (Optional) You can copy the data file from Amazon S3 to the local file system on the leader node. After you do this, you can use standard Linux command line utilities to work with the data in the file.

   ```
   aws s3 cp s3://aws-logs-123456789012-us-west-2/hive-test/000000_0 .
   ```

   The response should look similar to this:

   ```
   download: s3://aws-logs-123456789012-us-west-2/hive-test/000000_0
   to ./000000_0
   ```

   > **Note**
   >
   > The local file system on the leader node has limited capacity. Do not use this command with files that are larger than the available space in the local file system.

Copying data between DynamoDB and HDFS

If you have data in a DynamoDB table, you can use Hive to copy the data to the Hadoop Distributed File System (HDFS).

You might do this if you are running a MapReduce job that requires data from DynamoDB. If you copy the data from DynamoDB into HDFS, Hadoop can process it, using all of the available nodes in the Amazon EMR cluster in parallel. When the MapReduce job is complete, you can then write the results from HDFS to DDB.

In the following examples, Hive will read from and write to the following HDFS directory: `/user/hadoop/hive-test`

> **Note**
>
> The examples in this section are written with the assumption you followed the steps in Tutorial: Working with Amazon DynamoDB and Apache Hive and you have an external table in DynamoDB named *ddb_features*.

Topics

- Copying data using the Hive default format
- Copying data with a user-specified format
- Copying data without a column mapping
- Accessing the data in HDFS

Copying data using the Hive default format

Example From DynamoDB to HDFS

Use an `INSERT OVERWRITE` statement to write directly to HDFS.

```
INSERT OVERWRITE DIRECTORY 'hdfs:///user/hadoop/hive-test'
SELECT * FROM ddb_features;
```

The data file in HDFS looks like this:

```
920709^ASoldiers Farewell Hill^ASummit^ANM^A32.3564729^A-108.33004616135
```

```
1178153^AJones Run^AStream^APA^A41.2120086^A-79.25920781260
253838^ASentinel Dome^ASummit^ACA^A37.7229821^A-119.584338133
264054^ANeversweet Gulch^AValley^ACA^A41.6565269^A-122.83614322900
115905^AChacaloochee Bay^ABay^AAL^A30.6979676^A-87.97388530
```

Each field is separated by an SOH character (start of heading, 0x01). In the file, SOH appears as **^A**.

Example From HDFS to DynamoDB

1. Create an external table that maps to the unformatted data in HDFS.

   ```
   CREATE EXTERNAL TABLE hdfs_features_unformatted
       (feature_id       BIGINT,
       feature_name      STRING ,
       feature_class     STRING ,
       state_alpha       STRING,
       prim_lat_dec      DOUBLE ,
       prim_long_dec     DOUBLE ,
       elev_in_ft        BIGINT)
   LOCATION 'hdfs:///user/hadoop/hive-test';
   ```

2. Copy the data to DynamoDB.

   ```
   INSERT OVERWRITE TABLE ddb_features
   SELECT * FROM hdfs_features_unformatted;
   ```

Copying data with a user-specified format

If you want to use a different field separator character, you can create an external table that maps to the HDFS directory. You might use this technique for creating data files with comma-separated values (CSV).

Example From DynamoDB to HDFS

1. Create a Hive external table that maps to HDFS. When you do this, ensure that the data types are consistent with those of the DynamoDB external table.

   ```
   CREATE EXTERNAL TABLE hdfs_features_csv
       (feature_id       BIGINT,
       feature_name      STRING ,
       feature_class     STRING ,
   ```

```
    state_alpha       STRING,
    prim_lat_dec      DOUBLE ,
    prim_long_dec     DOUBLE ,
    elev_in_ft        BIGINT)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
LOCATION 'hdfs:///user/hadoop/hive-test';
```

2. Copy the data from DynamoDB.

```
INSERT OVERWRITE TABLE hdfs_features_csv
SELECT * FROM ddb_features;
```

The data file in HDFS looks like this:

```
920709,Soldiers Farewell Hill,Summit,NM,32.3564729,-108.3300461,6135
1178153,Jones Run,Stream,PA,41.2120086,-79.2592078,1260
253838,Sentinel Dome,Summit,CA,37.7229821,-119.58433,8133
264054,Neversweet Gulch,Valley,CA,41.6565269,-122.8361432,2900
115905,Chacaloochee Bay,Bay,AL,30.6979676,-87.9738853,0
```

Example From HDFS to DynamoDB

With a single HiveQL statement, you can populate the DynamoDB table using the data from HDFS:

```
INSERT OVERWRITE TABLE ddb_features
SELECT * FROM hdfs_features_csv;
```

Copying data without a column mapping

You can copy data from DynamoDB in a raw format and write it to HDFS without specifying any data types or column mapping. You can use this method to create an archive of DynamoDB data and store it in HDFS.

> **Note**
>
> If your DynamoDB table contains attributes of type Map, List, Boolean or Null, then this is the only way you can use Hive to copy data from DynamoDB to HDFS.

Example From DynamoDB to HDFS

1. Create an external table associated with your DynamoDB table. (There is no `dynamodb.column.mapping` in this HiveQL statement.)

```
CREATE EXTERNAL TABLE ddb_features_no_mapping
    (item MAP<STRING, STRING>)
STORED BY 'org.apache.hadoop.hive.dynamodb.DynamoDBStorageHandler'
TBLPROPERTIES ("dynamodb.table.name" = "Features");
```

2. Create another external table associated with your HDFS directory.

```
CREATE EXTERNAL TABLE hdfs_features_no_mapping
    (item MAP<STRING, STRING>)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\t'
LINES TERMINATED BY '\n'
LOCATION 'hdfs:///user/hadoop/hive-test';
```

3. Copy the data from DynamoDB to HDFS.

```
INSERT OVERWRITE TABLE hdfs_features_no_mapping
SELECT * FROM ddb_features_no_mapping;
```

The data file in HDFS looks like this:

```
Name^C{"s":"Soldiers Farewell
 Hill"}^BState^C{"s":"NM"}^BClass^C{"s":"Summit"}^BElevation^C{"n":"6135"}^BLatitude^C{"n":"32.
Name^C{"s":"Jones
 Run"}^BState^C{"s":"PA"}^BClass^C{"s":"Stream"}^BElevation^C{"n":"1260"}^BLatitude^C{"n":"41.2
Name^C{"s":"Sentinel
 Dome"}^BState^C{"s":"CA"}^BClass^C{"s":"Summit"}^BElevation^C{"n":"8133"}^BLatitude^C{"n":"37.
Name^C{"s":"Neversweet
 Gulch"}^BState^C{"s":"CA"}^BClass^C{"s":"Valley"}^BElevation^C{"n":"2900"}^BLatitude^C{"n":"41
Name^C{"s":"Chacaloochee
 Bay"}^BState^C{"s":"AL"}^BClass^C{"s":"Bay"}^BElevation^C{"n":"0"}^BLatitude^C{"n":"30.6979676
```

Each field begins with an STX character (start of text, 0x02) and ends with an ETX character (end of text, 0x03). In the file, STX appears as **^B** and ETX appears as **^C**.

Example From HDFS to DynamoDB

With a single HiveQL statement, you can populate the DynamoDB table using the data from HDFS:

```
INSERT OVERWRITE TABLE ddb_features_no_mapping
SELECT * FROM hdfs_features_no_mapping;
```

Accessing the data in HDFS

HDFS is a distributed file system, accessible to all of the nodes in the Amazon EMR cluster. If you use SSH to connect to the leader node, you can use command line tools to access the data that Hive wrote to HDFS.

HDFS is not the same thing as the local file system on the leader node. You cannot work with files and directories in HDFS using standard Linux commands (such as `cat`, `cp`, `mv`, or `rm`). Instead, you perform these tasks using the `hadoop fs` command.

The following steps are written with the assumption you have copied data from DynamoDB to HDFS using one of the procedures in this section.

1. If you are currently at the Hive command prompt, exit to the Linux command prompt.

   ```
   hive> exit;
   ```

2. List the contents of the /user/hadoop/hive-test directory in HDFS. (This is where Hive copied the data from DynamoDB.)

   ```
   hadoop fs -ls /user/hadoop/hive-test
   ```

   The response should look similar to this:

   ```
   Found 1 items
   -rw-r--r-- 1 hadoop hadoop 29504 2016-06-08 23:40 /user/hadoop/hive-test/000000_0
   ```

   The file name (*000000_0*) is system-generated.

3. View the contents of the file:

   ```
   hadoop fs -cat /user/hadoop/hive-test/000000_0
   ```

> **Note**
>
> In this example, the file is relatively small (approximately 29 KB). Be careful when you use this command with files that are very large or contain non-printable characters.

4. (Optional) You can copy the data file from HDFS to the local file system on the leader node. After you do this, you can use standard Linux command line utilities to work with the data in the file.

```
hadoop fs -get /user/hadoop/hive-test/000000_0
```

This command will not overwrite the file.

> **Note**
>
> The local file system on the leader node has limited capacity. Do not use this command with files that are larger than the available space in the local file system.

Using data compression

When you use Hive to copy data among different data sources, you can request on-the-fly data compression. Hive provides several compression codecs. You can choose one during your Hive session. When you do this, the data is compressed in the specified format.

The following example compresses data using the Lempel-Ziv-Oberhumer (LZO) algorithm.

```
SET hive.exec.compress.output=true;
SET io.seqfile.compression.type=BLOCK;
SET mapred.output.compression.codec = com.hadoop.compression.lzo.LzopCodec;

CREATE EXTERNAL TABLE lzo_compression_table (line STRING)
ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t' LINES TERMINATED BY '\n'
LOCATION 's3://bucketname/path/subpath/';

INSERT OVERWRITE TABLE lzo_compression_table SELECT *
FROM hiveTableName;
```

The resulting file in Amazon S3 will have a system-generated name with `.lzo` at the end (for example, `8d436957-57ba-4af7-840c-96c2fc7bb6f5-000000.lzo`).

The available compression codecs are:

- `org.apache.hadoop.io.compress.GzipCodec`
- `org.apache.hadoop.io.compress.DefaultCodec`
- `com.hadoop.compression.lzo.LzoCodec`
- `com.hadoop.compression.lzo.LzopCodec`
- `org.apache.hadoop.io.compress.BZip2Codec`
- `org.apache.hadoop.io.compress.SnappyCodec`

Reading non-printable UTF-8 character data

To read and write non-printable UTF-8 character data, you can use the `STORED AS SEQUENCEFILE` clause when you create a Hive table. A SequenceFile is a Hadoop binary file format. You need to use Hadoop to read this file. The following example shows how to export data from DynamoDB into Amazon S3. You can use this functionality to handle non-printable UTF-8 encoded characters.

```
CREATE EXTERNAL TABLE s3_export(a_col string, b_col bigint, c_col array<string>)
STORED AS SEQUENCEFILE
LOCATION 's3://bucketname/path/subpath/';

INSERT OVERWRITE TABLE s3_export SELECT *
FROM hiveTableName;
```

Performance tuning

When you create a Hive external table that maps to a DynamoDB table, you do not consume any read or write capacity from DynamoDB. However, read and write activity on the Hive table (such as `INSERT` or `SELECT`) translates directly into read and write operations on the underlying DynamoDB table.

Apache Hive on Amazon EMR implements its own logic for balancing the I/O load on the DynamoDB table and seeks to minimize the possibility of exceeding the table's provisioned throughput. At the end of each Hive query, Amazon EMR returns runtime metrics, including

the number of times your provisioned throughput was exceeded. You can use this information, together with CloudWatch metrics on your DynamoDB table, to improve performance in subsequent requests.

The Amazon EMR console provides basic monitoring tools for your cluster. For more information, see View and Monitor a Cluster in the *Amazon EMR Management Guide*.

You can also monitor your cluster and Hadoop jobs using web-based tools, such as Hue, Ganglia, and the Hadoop web interface. For more information, see View Web Interfaces Hosted on Amazon EMR Clusters in the *Amazon EMR Management Guide*.

This section describes steps you can take to performance-tune Hive operations on external DynamoDB tables.

Topics

- DynamoDB provisioned throughput
- Adjusting the mappers
- Additional topics

DynamoDB provisioned throughput

When you issue HiveQL statements against the external DynamoDB table, the `DynamoDBStorageHandler` class makes the appropriate low-level DynamoDB API requests, which consume provisioned throughput. If there is not enough read or write capacity on the DynamoDB table, the request will be throttled, resulting in slow HiveQL performance. For this reason, you should ensure that the table has enough throughput capacity.

For example, suppose that you have provisioned 100 read capacity units for your DynamoDB table. This will let you read 409,600 bytes per second (100 × 4 KB read capacity unit size). Now suppose that the table contains 20 GB of data (21,474,836,480 bytes) and you want to use the `SELECT` statement to select all of the data using HiveQL. You can estimate how long the query will take to run like this:

21,474,836,480 / 409,600 = 52,429 seconds = 14.56 hours

In this scenario, the DynamoDB table is a bottleneck. It won't help to add more Amazon EMR nodes, because the Hive throughput is constrained to only 409,600 bytes per second. The only way to decrease the time required for the `SELECT` statement is to increase the provisioned read capacity of the DynamoDB table.

You can perform a similar calculation to estimate how long it would take to bulk-load data into a Hive external table mapped to a DynamoDB table. Determine the total number of write capacity units needed per item (less than 1KB = 1, 1-2KB = 2, etc), and multiply that by the number of items to load. This will give you the number of write capacity units required. Divide that number by the number of write capacity units that are allocated per second. This will yield the number of seconds it will take to load the table.

You should regularly monitor the CloudWatch metrics for your table. For a quick overview in the DynamoDB console, choose your table and then choose the **Metrics** tab. From here, you can view read and write capacity units consumed and read and write requests that have been throttled.

Read capacity

Amazon EMR manages the request load against your DynamoDB table, according to the table's provisioned throughput settings. However, if you notice a large number of `ProvisionedThroughputExceeded` messages in the job output, you can adjust the default read rate. To do this, you can modify the `dynamodb.throughput.read.percent` configuration variable. You can use the `SET` command to set this variable at the Hive command prompt:

```
SET dynamodb.throughput.read.percent=1.0;
```

This variable persists for the current Hive session only. If you exit Hive and return to it later, `dynamodb.throughput.read.percent` will return to its default value.

The value of `dynamodb.throughput.read.percent` can be between `0.1` and `1.5`, inclusively. `0.5` represents the default read rate, meaning that Hive will attempt to consume half of the read capacity of the table. If you increase the value above `0.5`, Hive will increase the request rate; decreasing the value below `0.5` decreases the read request rate. (The actual read rate will vary, depending on factors such as whether there is a uniform key distribution in the DynamoDB table.)

If you notice that Hive is frequently depleting the provisioned read capacity of the table, or if your read requests are being throttled too much, try reducing `dynamodb.throughput.read.percent` below `0.5`. If you have sufficient read capacity in the table and want more responsive HiveQL operations, you can set the value above `0.5`.

Write capacity

Amazon EMR manages the request load against your DynamoDB table, according to the table's provisioned throughput settings. However, if you notice a large number of

`ProvisionedThroughputExceeded` messages in the job output, you can adjust the default write rate. To do this, you can modify the `dynamodb.throughput.write.percent` configuration variable. You can use the `SET` command to set this variable at the Hive command prompt:

```
SET dynamodb.throughput.write.percent=1.0;
```

This variable persists for the current Hive session only. If you exit Hive and return to it later, `dynamodb.throughput.write.percent` will return to its default value.

The value of `dynamodb.throughput.write.percent` can be between `0.1` and `1.5`, inclusively. `0.5` represents the default write rate, meaning that Hive will attempt to consume half of the write capacity of the table. If you increase the value above `0.5`, Hive will increase the request rate; decreasing the value below `0.5` decreases the write request rate. (The actual write rate will vary, depending on factors such as whether there is a uniform key distribution in the DynamoDB table.)

If you notice that Hive is frequently depleting the provisioned write capacity of the table, or if your write requests are being throttled too much, try reducing `dynamodb.throughput.write.percent` below `0.5`. If you have sufficient capacity in the table and want more responsive HiveQL operations, you can set the value above `0.5`.

When you write data to DynamoDB using Hive, ensure that the number of write capacity units is greater than the number of mappers in the cluster. For example, consider an Amazon EMR cluster consisting of 10 *m1.xlarge* nodes. The *m1.xlarge* node type provides 8 mapper tasks, so the cluster would have a total of 80 mappers (10 × 8). If your DynamoDB table has fewer than 80 write capacity units, then a Hive write operation could consume all of the write throughput for that table.

To determine the number of mappers for Amazon EMR node types, see Task Configuration in the *Amazon EMR Developer Guide*.

For more information on mappers, see Adjusting the mappers.

Adjusting the mappers

When Hive launches a Hadoop job, the job is processed by one or more mapper tasks. Assuming that your DynamoDB table has sufficient throughput capacity, you can modify the number of mappers in the cluster, potentially improving performance.

> **Note**
>
> The number of mapper tasks used in a Hadoop job are influenced by *input splits*, where Hadoop subdivides the data into logical blocks. If Hadoop does not perform enough input splits, then your write operations might not be able to consume all the write throughput available in the DynamoDB table.

Increasing the number of mappers

Each mapper in an Amazon EMR has a maximum read rate of 1 MiB per second. The number of mappers in a cluster depends on the size of the nodes in your cluster. (For information about node sizes and the number of mappers per node, see Task Configuration in the *Amazon EMR Developer Guide*.)

If your DynamoDB table has ample throughput capacity for reads, you can try increasing the number of mappers by doing one of the following:

- Increase the size of the nodes in your cluster. For example, if your cluster is using *m1.large* nodes (three mappers per node), you can try upgrading to *m1.xlarge* nodes (eight mappers per node).
- Increase the number of nodes in your cluster. For example, if you have three-node cluster of *m1.xlarge* nodes, you have a total of 24 mappers available. If you were to double the size of the cluster, with the same type of node, you would have 48 mappers.

You can use the AWS Management Console to manage the size or the number of nodes in your cluster. (You might need to restart the cluster for these changes to take effect.)

Another way to increase the number of mappers is to modify the `mapred.tasktracker.map.tasks.maximum` Hadoop configuration parameter. (This is a Hadoop parameter, not a Hive parameter. You cannot modify it interactively from the command prompt.). If you increase the value of `mapred.tasktracker.map.tasks.maximum`, you can increase the number of mappers without increasing the size or number of nodes. However, it is possible for the cluster nodes to run out of memory if you set the value too high.

You set the value for `mapred.tasktracker.map.tasks.maximum` as a bootstrap action when you first launch your Amazon EMR cluster. For more information, see (Optional) Create Bootstrap Actions to Install Additional Software in the *Amazon EMR Management Guide*.

Decreasing the number of mappers

If you use the SELECT statement to select data from an external Hive table that maps to DynamoDB, the Hadoop job can use as many tasks as necessary, up to the maximum number of mappers in the cluster. In this scenario, it is possible that a long-running Hive query can consume all of the provisioned read capacity of the DynamoDB table, negatively impacting other users.

You can use the `dynamodb.max.map.tasks` parameter to set an upper limit for map tasks:

```
SET dynamodb.max.map.tasks=1
```

This value must be equal to or greater than 1. When Hive processes your query, the resulting Hadoop job will use no more than `dynamodb.max.map.tasks` when reading from the DynamoDB table.

Additional topics

The following are some more ways to tune applications that use Hive to access DynamoDB.

Retry duration

By default, Hive will rerun a Hadoop job if it has not returned any results from DynamoDB within two minutes. You can adjust this interval by modifying the `dynamodb.retry.duration` parameter:

```
SET dynamodb.retry.duration=2;
```

The value must be a nonzero integer, representing the number of minutes in the retry interval. The default for `dynamodb.retry.duration` is 2 (minutes).

Parallel data requests

Multiple data requests, either from more than one user or more than one application to a single table can drain read provisioned throughput and slow performance.

Process duration

Data consistency in DynamoDB depends on the order of read and write operations on each node. While a Hive query is in progress, another application might load new data into the DynamoDB

table or modify or delete existing data. In this case, the results of the Hive query might not reflect changes made to the data while the query was running.

Request time

Scheduling Hive queries that access a DynamoDB table when there is lower demand on the DynamoDB table improves performance. For example, if most of your application's users live in San Francisco, you might choose to export daily data at 4:00 A.M. PST when the majority of users are asleep and not updating records in your DynamoDB database.

Integrating DynamoDB with Amazon S3

Amazon DynamoDB import and export capabilities provide a simple and efficient way to move data between Amazon S3 and DynamoDB tables without writing any code.

DynamoDB import and export features help you move, transform, and copy DynamoDB table accounts. You can import from your S3 sources, and you can export your DynamoDB table data to Amazon S3 and use AWS services such as Athena, Amazon SageMaker AI, and AWS Lake Formation to analyze your data and extract actionable insights. You can also import data directly into new DynamoDB tables to build new applications with single-digit millisecond performance at scale, facilitate data sharing between tables and accounts, and simplify your disaster recovery and business continuity plans.

Topics

- DynamoDB data import from Amazon S3: how it works
- DynamoDB data export to Amazon S3: how it works

DynamoDB data import from Amazon S3: how it works

To import data into DynamoDB, your data must be in an Amazon S3 bucket in CSV, DynamoDB JSON, or Amazon Ion format. Data can be compressed in ZSTD or GZIP format, or can be directly imported in uncompressed form. Source data can either be a single Amazon S3 object or multiple Amazon S3 objects that use the same prefix.

Your data will be imported into a new DynamoDB table, which will be created when you initiate the import request. You can create this table with secondary indexes, then query and update your

data across all primary and secondary indexes as soon as the import is complete. You can also add a global table replica after the import is complete.

> **Note**
>
> During the Amazon S3 import process, DynamoDB creates a new target table that will be imported into. Import into existing tables is not currently supported by this feature.

Import from Amazon S3 does not consume write capacity on the new table, so you do not need to provision any extra capacity for importing data into DynamoDB. Data import pricing is based on the uncompressed size of the source data in Amazon S3, that is processed as a result of the import. Items that are processed but fail to load into the table due to formatting or other inconsistencies in the source data are also billed as part of the import process. See Amazon DynamoDB pricing for details.

You can import data from an Amazon S3 bucket owned by a different account if you have the correct permissions to read from that specific bucket. The new table may also be in a different Region from the source Amazon S3 bucket. For more information, see Amazon Simple Storage Service setup and permissions .

Import times are directly related to your data's characteristics in Amazon S3. This includes data size, data format, compression scheme, uniformity of data distribution, number of Amazon S3 objects, and other related variables. In particular, data sets with uniformly distributed keys will be faster to import than skewed data sets. For example, if your secondary index's key is using the month of the year for partitioning, and all your data is from the month of December, then importing this data may take significantly longer.

The attributes associated with keys are expected to be unique on the base table. If any keys are not unique, the import will overwrite the associated items until only the last overwrite remains. For example, if the primary key is the month and multiple items are set to the month of September, each new item will overwrite the previously written items and only one item with the primary key of "month" set to September will remain. In such cases, the number of items processed in the import table description will not match the number of items in the target table.

AWS CloudTrail logs all console and API actions for table import. For more information, see Logging DynamoDB operations by using AWS CloudTrail.

The following video is an introduction to importing directly from Amazon S3 into DynamoDB.

Import from Amazon S3

Topics

- Requesting a table import in DynamoDB
- Amazon S3 import formats for DynamoDB
- Import format quotas and validation
- Best practices for importing from Amazon S3 into DynamoDB

Requesting a table import in DynamoDB

DynamoDB import allows you to import data from an Amazon S3 bucket to a new DynamoDB table. You can request a table import using the DynamoDB console, the CLI, CloudFormation or the DynamoDB API.

If you want to use the AWS CLI, you must configure it first. For more information, see Accessing DynamoDB.

Note

- The Import Table feature interacts with multiple different AWS Services such as Amazon S3 and CloudWatch. Before you begin an import, make sure that the user or role that invokes the import APIs has permissions to all services and resources the feature depends on.
- Do not modify the Amazon S3 objects while the import is in progress, as this can cause the operation to fail or be cancelled.

For more information on errors and troubleshooting, see Import format quotas and validation

Topics

- Setting up IAM permissions
- Requesting an import using the AWS Management Console
- Getting details about past imports in the AWS Management Console

- Requesting an import using the AWS CLI
- Getting details about past imports in the AWS CLI

Setting up IAM permissions

You can import data from any Amazon S3 bucket you have permission to read from. The source bucket does not need to be in the same Region or have the same owner as the source table. Your AWS Identity and Access Management (IAM) must include the relevant actions on the source Amazon S3 bucket, and required CloudWatch permissions for providing debugging information. An example policy is shown below.

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Sid": "AllowDynamoDBImportAction",
      "Effect": "Allow",
      "Action": [
        "dynamodb:ImportTable",
        "dynamodb:DescribeImport"
      ],
      "Resource": "arn:aws:dynamodb:us-east-1:111122223333:table/my-table*"
    },
    {
      "Sid": "AllowS3Access",
      "Effect": "Allow",
      "Action": [
        "s3:GetObject",
        "s3:ListBucket"
      ],
      "Resource": [
        "arn:aws:s3:::your-bucket/*",
        "arn:aws:s3:::your-bucket"
      ]
    },
    {
      "Sid": "AllowCloudwatchAccess",
      "Effect": "Allow",
      "Action": [
```

```
        "logs:CreateLogGroup",
        "logs:CreateLogStream",
        "logs:DescribeLogGroups",
        "logs:DescribeLogStreams",
        "logs:PutLogEvents",
        "logs:PutRetentionPolicy"
      ],
      "Resource": "arn:aws:logs:us-east-1:111122223333:log-group/aws-dynamodb/*"
    },
    {
      "Sid": "AllowDynamoDBListImports",
      "Effect": "Allow",
      "Action": "dynamodb:ListImports",
      "Resource": "*"
    }
  ]
}
```

Amazon S3 permissions

When starting an import on an Amazon S3 bucket source that is owned by another account, ensure that the role or user has access to the Amazon S3 objects. You can check that by executing an Amazon S3 `GetObject` command and using the credentials. When using the API, the Amazon S3 bucket owner parameter defaults to the current user's account ID. For cross account imports, ensure that this parameter is correctly populated with the bucket owner's account ID. The following code is an example Amazon S3 bucket policy in the source account.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {"Sid": "ExampleStatement",
            "Effect": "Allow",
            "Principal": {"AWS": "arn:aws:iam::123456789012:user/Dave"
            },
            "Action": [
                "s3:GetObject",
                "s3:ListBucket"
            ],
            "Resource": "arn:aws:s3:::amzn-s3-demo-bucket/*"
```

```
        }
    ]
}
```

AWS Key Management Service

When creating the new table for import, if you select an encryption at rest key that is not owned by DynamoDB then you must provide the AWS KMS permissions required to operate a DynamoDB table encrypted with customer managed keys. For more information see Authorizing use of your AWS KMS key. If the Amazon S3 objects are encrypted with server side encryption KMS (SSE-KMS), ensure that the role or user initiating the import has access to decrypt using the AWS KMS key. This feature does not support customer-provided encryption keys (SSE-C) encrypted Amazon S3 objects.

CloudWatch permissions

The role or user that is initiating the import will need create and manage permissions for the log group and log streams associated with the import.

Requesting an import using the AWS Management Console

The following example demonstrates how to use the DynamoDB console to import existing data to a new table named `MusicCollection`.

To request a table import

1. Sign in to the AWS Management Console and open the DynamoDB console at https://console.aws.amazon.com/dynamodb/.
2. In the navigation pane on the left side of the console, choose **Import from S3**.
3. On the page that appears, select **Import from S3**.
4. Choose **Import from S3**.
5. In **Source S3 URL**, enter the Amazon S3 source URL.

   If you own the source bucket, choose **Browse S3** to search for it. Alternatively, enter the bucket's URL in the following format – `s3://bucket/prefix`. The `prefix` is an Amazon S3 key prefix. It's either the Amazon S3 object name that you want to import or the key prefix shared by all the Amazon S3 objects that you want to import.

> **Note**
>
> You can't use the same prefix as your DynamoDB export request. The export feature creates a folder structure and manifest files for all the exports. If you use the same Amazon S3 path, it will result in an error.
> Instead, point the import at the folder, which contains data from that specific export. The format of the correct path in this case will be `s3://bucket/prefix/AWSDynamoDB/<XXXXXXXX-XXXXXX>/data/`, where `XXXXXXXX-XXXXXX` is the export ID. You can find export ID in the export ARN, which has the following format – `arn:aws:dynamodb:<Region>:<AccountID>:table/<TableName>/export/<XXXXXXXX-XXXXXX>`. For example, `arn:aws:dynamodb:us-east-1:123456789012:table/ProductCatalog/export/01234567890123-a1b2c3d4`.

6. Specify if you are the **S3 bucket owner**. If the source bucket is owned by a different account, select **A different AWS account**. Then enter the account ID of the bucket owner.
7. Under **Import file compression**, select either **No compression**, **GZIP** or **ZSTD** as appropriate.
8. Select the appropriate Import file format. The options are **DynamoDB JSON**, **Amazon Ion** or **CSV**. If you select **CSV**, you will have two additional options: **CSV header** and **CSV delimiter character**.

   For **CSV header**, choose if the header will either be taken from the first line of the file or be customized. If you select **Customize your headers**, you can specify the header values you want to import with. CSV Headers specified by this method are case-sensitive and are expected to contain the keys of the target table.

   For **CSV delimiter character**, you set the character which will separate items. Comma is selected by default. If you select **Custom delimiter character**, the delimiter must match the regex pattern: `[,;:|\t ]`.
9. Select the **Next** button and select the options for the new table that will be created to store your data.

> **Note**
>
> Primary Key and Sort Key must match the attributes in the file, or the import will fail. The attributes are case sensitive.

10. Select **Next** again to review your import options, then click **Import** to begin the import task. You will first see your new table listed in the “Tables” with the status “Creating”. At this time the table is not accessible.
11. Once the import completes, the status will show as "Active" and you can start using the table.

Getting details about past imports in the AWS Management Console

You can find information about import tasks you've run in the past by clicking **Import from S3** in the navigation sidebar, then selecting the **Imports** tab. The import panel contains a list of all imports you've created in the past 90 days. Selecting the ARN of a task listed in the Imports tab will retrieve information about that import, including any advanced configuration settings you chose.

Requesting an import using the AWS CLI

The following example imports CSV formatted data from an S3 bucket called bucket with a prefix of prefix to a new table called target-table.

```
aws dynamodb import-table --s3-bucket-source S3Bucket=bucket,S3KeyPrefix=prefix \
            --input-format CSV --table-creation-parameters '{"TableName":"target-
table","KeySchema":  \
            [{"AttributeName":"hk","KeyType":"HASH"}],"AttributeDefinitions":
[{"AttributeName":"hk","AttributeType":"S"}],"BillingMode":"PAY_PER_REQUEST"}' \
            --input-format-options '{"Csv": {"HeaderList": ["hk", "title", "artist",
 "year_of_release"], "Delimiter": ";"}}'
```

> **Note**
>
> If you choose to encrypt your import using a key protected by AWS Key Management Service (AWS KMS), the key must be in the same Region as the destination Amazon S3 bucket.

Getting details about past imports in the AWS CLI

You can find information about import tasks you've run in the past by using the `list-imports` command. This command returns a list of all imports you've created in the past 90 days. Note that although import task metadata expires after 90 days and jobs older than that are no longer found

on this list, DynamoDB does not delete any of the objects in your Amazon S3 bucket or the table created during import.

```
aws dynamodb list-imports
```

To retrieve detailed information about a specific import task, including any advanced configuration settings, use the `describe-import` command.

```
aws dynamodb describe-import \
    --import-arn arn:aws:dynamodb:us-east-1:123456789012:table/ProductCatalog/exp
```

Amazon S3 import formats for DynamoDB

DynamoDB can import data in three formats: CSV, DynamoDB JSON, and Amazon Ion.

Topics

- CSV
- DynamoDB Json
- Amazon Ion

CSV

A file in CSV format consists of multiple items delimited by newlines. By default, DynamoDB interprets the first line of an import file as the header and expects columns to be delimited by commas. You can also define headers that will be applied, as long as they match the number of columns in the file. If you define headers explicitly, the first line of the file will be imported as values.

> **Note**
>
> When importing from CSV files, all columns other than the hash range and keys of your base table and secondary indexes are imported as DynamoDB strings.

Escaping double quotes

Any double quotes characters that exist in the CSV file must be escaped. If they are not escaped, such as in this following example, the import will fail:

```
id,value
"123",Women's Full "Length" Dress
```

This same import will succeed if the quotes are escaped with two sets of double quotes:

```
id,value
"""123""","Women's Full ""Length"" Dress"
```

Once the text has been properly escaped and imported, it will appear as it did in the original CSV file:

```
id,value
"123",Women's Full "Length" Dress
```

Importing heterogeneous item types

You can use a single CSV file to import different item types into one table. Define a header row that includes all attributes across your item types, and leave columns empty for attributes that don't apply to a given item. Empty columns are omitted from the imported item rather than stored as empty strings.

```
PK,SK,EntityType,Name,Email,OrderDate,Amount,ProductName,Quantity
USER#1,PROFILE,User,Alice,alice@example.com,,,,
USER#1,ORDER#2024-01-15,Order,,,2024-01-15,99.99,,
USER#1,ORDER#2024-02-10,Order,,,2024-02-10,149.50,,
PRODUCT#101,METADATA,Product,,,,,Laptop,50
PRODUCT#102,METADATA,Product,,,,,Mouse,200
USER#2,PROFILE,User,Bob,bob@example.com,,,,
USER#2,ORDER#2024-01-20,Order,,,2024-01-20,75.00,,
PRODUCT#103,METADATA,Product,,,,,Keyboard,150
USER#3,PROFILE,User,Charlie,charlie@example.com,,,,
PRODUCT#104,METADATA,Product,,,,,Monitor,30
```

In this example, user profiles, orders, and products share the same table. Each item type uses only the columns relevant to it.

DynamoDB Json

A file in DynamoDB JSON format can consist of multiple Item objects. Each individual object is in DynamoDB's standard marshalled JSON format, and newlines are used as item delimiters. As an added feature, exports from point in time are supported as an import source by default.

> **Note**
>
> New lines are used as item delimiters for a file in DynamoDB JSON format and shouldn't be used within an item object.

```
{"Item": {"Authors": {"SS": ["Author1", "Author2"]}, "Dimensions": {"S": "8.5 x 11.0 x
 1.5"}, "ISBN": {"S": "333-3333333333"}, "Id": {"N": "103"}, "InPublication": {"BOOL":
 false}, "PageCount": {"N": "600"}, "Price": {"N": "2000"}, "ProductCategory": {"S":
 "Book"}, "Title": {"S": "Book 103 Title"}}}
{"Item": {"Authors": {"SS": ["Author1", "Author2"]}, "Dimensions": {"S": "8.5 x 11.0 x
 1.5"}, "ISBN": {"S": "444-444444444"}, "Id": {"N": "104"}, "InPublication": {"BOOL":
 false}, "PageCount": {"N": "600"}, "Price": {"N": "2000"}, "ProductCategory": {"S":
 "Book"}, "Title": {"S": "Book 104 Title"}}}
{"Item": {"Authors": {"SS": ["Author1", "Author2"]}, "Dimensions": {"S": "8.5 x 11.0 x
 1.5"}, "ISBN": {"S": "555-5555555555"}, "Id": {"N": "105"}, "InPublication": {"BOOL":
 false}, "PageCount": {"N": "600"}, "Price": {"N": "2000"}, "ProductCategory": {"S":
 "Book"}, "Title": {"S": "Book 105 Title"}}}
```

Amazon Ion

Amazon Ion is a richly-typed, self-describing, hierarchical data serialization format built to address rapid development, decoupling, and efficiency challenges faced every day while engineering large-scale, service-oriented architectures.

When you import data in Ion format, the Ion datatypes are mapped to DynamoDB datatypes in the new DynamoDB table.

S. No.	Ion to DynamoDB datatype conversion	B
1	Ion Data Type	DynamoDB Representation
2	string	String (s)
3	bool	Boolean (BOOL)
4	decimal	Number (N)

S. No.	Ion to DynamoDB datatype conversion	B
5	blob	Binary (B)
6	list (with type annotation $dynamodb_SS, $dynamodb_NS, or $dynamodb_BS)	Set (SS, NS, BS)
7	list	List
8	struct	Map

Items in an Ion file are delimited by newlines. Each line begins with an Ion version marker, followed by an item in Ion format.

> **Note**
>
> In the following example, we've formatted items from an Ion-formatted file on multiple lines to improve readability.

```
$ion_1_0
[
  {
    Item:{
      Authors:$dynamodb_SS::["Author1","Author2"],
      Dimensions:"8.5 x 11.0 x 1.5",
      ISBN:"333-3333333333",
      Id:103.,
      InPublication:false,
      PageCount:6d2,
      Price:2d3,
      ProductCategory:"Book",
      Title:"Book 103 Title"
    }
  },
  {
    Item:{
```

```
      Authors:$dynamodb_SS::["Author1","Author2"],
      Dimensions:"8.5 x 11.0 x 1.5",
      ISBN:"444-4444444444",
      Id:104.,
      InPublication:false,
      PageCount:6d2,
      Price:2d3,
      ProductCategory:"Book",
      Title:"Book 104 Title"
    }
  },
  {
    Item:{
      Authors:$dynamodb_SS::["Author1","Author2"],
      Dimensions:"8.5 x 11.0 x 1.5",
      ISBN:"555-5555555555",
      Id:105.,
      InPublication:false,
      PageCount:6d2,
      Price:2d3,
      ProductCategory:"Book",
      Title:"Book 105 Title"
    }
  }
]
```

Import format quotas and validation

Import quotas

DynamoDB Import from Amazon S3 can support up to 50 concurrent import jobs with a total import source object size of 15TB at a time in us-east-1, us-west-2, and eu-west-1 regions. In all other regions, up to 50 concurrent import tasks with a total size of 1TB is supported. Each import job can take up to 50,000 Amazon S3 objects in all regions. These default quotas are applied to every account. If you feel you need to revise these quotas, please contact your account team, and this will be considered on a case-by-case basis. For more details on DynamoDB limits, see Service Quotas.

Validation errors

During the import process, DynamoDB may encounter errors while parsing your data. For each error, DynamoDB emits a CloudWatch log and keeps a count of the total number of errors

encountered. If the Amazon S3 object itself is malformed or if its contents cannot form a DynamoDB item, then we may skip processing the remaining portion of the object.

> **Note**
>
> If the Amazon S3 data source has multiple items that share the same key, the items will overwrite until one remains. This can appear as if 1 item was imported and the others were ignored. The duplicate items will be overwritten in random order, are not counted as errors, and are not emitted to CloudWatch logs.
> Once the import is complete you can see the total count of items imported, total count of errors, and total count of items processed. For further troubleshooting you can also check the total size of items imported and total size of data processed.

There are three categories of import errors: API validation errors, data validation errors, and configuration errors.

API validation errors

API validation errors are item-level errors from the sync API. Common causes are permissions issues, missing required parameters and parameter validation failures. Details on why the API call failed are contained in the exceptions thrown by the `ImportTable` request.

Data validation errors

Data validation errors can occur at either the item level or file level. During import, items are validated based on DynamoDB rules before importing into the target table. When an item fails validation and is not imported, the import job skips over that item and continues on with the next item. At the end of job, the import status is set to FAILED with a FailureCode, ItemValidationError and the FailureMessage "Some of the items failed validation checks and were not imported. Please check CloudWatch error logs for more details."

Common causes for data validation errors include objects being unparsable, objects being in the incorrect format (input specifies DYNAMODB_JSON but the object is not in DYNAMODB_JSON), and schema mismatch with specified source table keys.

Configuration errors

Configuration errors are typically workflow errors due to permission validation. The Import workflow checks some permissions after accepting the request. If there are issues calling any of

the required dependencies like Amazon S3 or CloudWatch the process marks the import status as FAILED. The `failureCode` and `failureMessage` point to the reason for failure. Where applicable, the failure message also contains the request id that you can use to investigate the reason for failure in CloudTrail.

Common configuration errors include having the wrong URL for the Amazon S3 bucket, and not having permission to access the Amazon S3 bucket, CloudWatch Logs, and AWS KMS keys used to decrypt the Amazon S3 object. For more information see Using and data keys.

Validating source Amazon S3 objects

In order to validate source S3 objects, take the following steps.

1. Validate the data format and compression type

   - Make sure that all matching Amazon S3 objects under the specified prefix have the same format (DYNAMODB_JSON, DYNAMODB_ION, CSV)
   - Make sure that all matching Amazon S3 objects under the specified prefix are compressed the same way (GZIP, ZSTD, NONE)

   > **Note**
   >
   > The Amazon S3 objects do not need to have the corresponding extension (.csv / .json / .ion / .gz / .zstd etc) as the input format specified in ImportTable call takes precedence.

2. Validate that the import data conforms to the desired table schema

   - Make sure that each item in the source data has the primary key. A sort key is optional for imports.
   - Make sure that the attribute type associated with the primary key and any sort key matches the attribute type in the Table and the GSI schema, as specified in table creation parameters

Troubleshooting

CloudWatch logs

For Import jobs that fail, detailed error messages are posted to CloudWatch logs. To access these logs, first retrieve the ImportArn from the output and describe-import using this command:

```
aws dynamodb describe-import --import-arn arn:aws:dynamodb:us-east-1:ACCOUNT:table/
target-table/import/01658528578619-c4d4e311
}
```

Example output:

```
aws dynamodb describe-import --import-arn "arn:aws:dynamodb:us-
east-1:531234567890:table/target-table/import/01658528578619-c4d4e311"
{
    "ImportTableDescription": {
        "ImportArn": "arn:aws:dynamodb:us-east-1:ACCOUNT:table/target-table/
import/01658528578619-c4d4e311",
        "ImportStatus": "FAILED",
        "TableArn": "arn:aws:dynamodb:us-east-1:ACCOUNT:table/target-table",
        "TableId": "7b7ecc22-302f-4039-8ea9-8e7c3eb2bcb8",
        "ClientToken": "30f8891c-e478-47f4-af4a-67a5c3b595e3",
        "S3BucketSource": {
            "S3BucketOwner": "ACCOUNT",
            "S3Bucket": "my-import-source",
            "S3KeyPrefix": "import-test"
        },
        "ErrorCount": 1,
        "CloudWatchLogGroupArn": "arn:aws:logs:us-east-1:ACCOUNT:log-group:/aws-
dynamodb/imports:*",
        "InputFormat": "CSV",
        "InputCompressionType": "NONE",
        "TableCreationParameters": {
            "TableName": "target-table",
            "AttributeDefinitions": [
                {
                    "AttributeName": "pk",
                    "AttributeType": "S"
                }
            ],
            "KeySchema": [
                {
                    "AttributeName": "pk",
                    "KeyType": "HASH"
                }
            ],
            "BillingMode": "PAY_PER_REQUEST"
        },
        "StartTime": 1658528578.619,
```

```
        "EndTime": 1658528750.628,
        "ProcessedSizeBytes": 70,
        "ProcessedItemCount": 1,
        "ImportedItemCount": 0,
        "FailureCode": "ItemValidationError",
        "FailureMessage": "Some of the items failed validation checks and were not
 imported. Please check CloudWatch error logs for more details."
    }
}
```

Retrieve the log group and the import id from the above response and use it to retrieve the error logs. The import ID is the last path element of the `ImportArn` field. The log group name is `/aws-dynamodb/imports`. The error log stream name is `import-id/error`. For this example, it would be `01658528578619-c4d4e311/error`.

Missing the key pk in the item

If the source S3 object does not contain the primary key that was provided as a parameter, the import will fail. For example, when you define the primary key for the import as column name “pk”.

```
aws dynamodb import-table –s3-bucket-source S3Bucket=my-import-
source,S3KeyPrefix=import-test.csv \
            –input-format CSV --table-creation-parameters '{"TableName":"target-
table","KeySchema":  \
            [{"AttributeName":"pk","KeyType":"HASH"}],"AttributeDefinitions":
[{"AttributeName":"pk","AttributeType":"S"}],"BillingMode":"PAY_PER_REQUEST"}'
```

The column “pk” is missing from the the source object `import-test.csv` which has the following contents:

```
title,artist,year_of_release
The Dark Side of the Moon,Pink Floyd,1973
```

This import will fail due to item validation error because of the missing primary key in the data source.

Example CloudWatch error log:

```
aws logs get-log-events –log-group-name /aws-dynamodb/imports –log-stream-name
 01658528578619-c4d4e311/error
{
"events": [
```

```
{
"timestamp": 1658528745319,
"message": "{\"itemS3Pointer\":{\"bucket\":\"my-import-source\",\"key\":
\"import-test.csv\",\"itemIndex\":0},\"importArn\":\"arn:aws:dynamodb:us-
east-1:531234567890:table/target-table/import/01658528578619-c4d4e311\",\"errorMessages
\":[\"One or more parameter values were invalid: Missing the key pk in the item\"]}",
"ingestionTime": 1658528745414
}
],
"nextForwardToken": "f/36986426953797707963335499204463414460239026137054642176/s",
"nextBackwardToken": "b/36986426953797707963335499204463414460239026137054642176/s"
}
```


This error log indicates that “One or more parameter values were invalid: Missing the key pk in the item”. Since this import job failed, the table “target-table” now exists and is empty because no items were imported. The first item was processed and the object failed Item Validation.

To fix the issue, first delete “target-table” if it is no longer needed. Then either use a primary key column name that exists in the source object, or update the source data to:

```
pk,title,artist,year_of_release
Albums::Rock::Classic::1973::AlbumId::ALB25,The Dark Side of the Moon,Pink Floyd,1973
```

Target table exists

When you start an import job and receive a response as follows:

```
An error occurred (ResourceInUseException) when calling the ImportTable operation:
 Table already exists: target-table
```

To fix this error, you will need to choose a table name that doesn’t already exist and retry the import.

The specified bucket does not exist

If the source bucket does not exist, the import will fail and log the error message details in CloudWatch.

Example describe import:

```
aws dynamodb –endpoint-url $ENDPOINT describe-import –import-arn "arn:aws:dynamodb:us-
east-1:531234567890:table/target-table/import/01658530687105-e6035287"
```

```
{
"ImportTableDescription": {
"ImportArn": "arn:aws:dynamodb:us-east-1:ACCOUNT:table/target-table/
import/01658530687105-e6035287",
"ImportStatus": "FAILED",
"TableArn": "arn:aws:dynamodb:us-east-1:ACCOUNT:table/target-table",
"TableId": "e1215a82-b8d1-45a8-b2e2-14b9dd8eb99c",
"ClientToken": "3048e16a-069b-47a6-9dfb-9c259fd2fb6f",
"S3BucketSource": {
"S3BucketOwner": "531234567890",
"S3Bucket": "BUCKET_DOES_NOT_EXIST",
"S3KeyPrefix": "import-test"
},
"ErrorCount": 0,
"CloudWatchLogGroupArn": "arn:aws:logs:us-east-1:ACCOUNT:log-group:/aws-dynamodb/
imports:*",
"InputFormat": "CSV",
"InputCompressionType": "NONE",
"TableCreationParameters": {
"TableName": "target-table",
"AttributeDefinitions": [
{
"AttributeName": "pk",
"AttributeType": "S"
}
],
"KeySchema": [
{
"AttributeName": "pk",
"KeyType": "HASH"
}
],
"BillingMode": "PAY_PER_REQUEST"
},
"StartTime": 1658530687.105,
"EndTime": 1658530701.873,
"ProcessedSizeBytes": 0,
"ProcessedItemCount": 0,
"ImportedItemCount": 0,
"FailureCode": "S3NoSuchBucket",
"FailureMessage": "The specified bucket does not exist (Service: Amazon S3; Status
 Code: 404; Error Code: NoSuchBucket; Request ID: Q4W6QYYFDWY6WAKH; S3 Extended Request
 ID: ObqSlLeIMJpQqHLRX2C5Sy7n+8g6iGPwy7ixg7eEeTuEkg/+chU/JF+RbliWytMlkUlUcuCLTrI=;
 Proxy: null)"
```

```
}
}
```

The `FailureCode` is `S3NoSuchBucket`, with `FailureMessage` containing details such as request id and the service that threw the error. Since the error was caught before the data was imported into the table, a new DynamoDB table is not created. In some cases, when these errors are encountered after the data import has started, the table with partially imported data is retained.

To fix this error, make sure that the source Amazon S3 bucket exists and then restart the import process.

Best practices for importing from Amazon S3 into DynamoDB

The following are the best practices for importing data from Amazon S3 into DynamoDB.

Stay under the limit of 50,000 S3 objects

Each import job supports a maximum of 50,000 S3 objects. If your dataset contains more than 50,000 objects, consider consolidating them into larger objects.

Avoid excessively large S3 objects

S3 objects are imported in parallel. Having numerous mid-sized S3 objects allows for parallel execution without excessive overhead. For items under 1 KB, consider placing 4,000,000 items into each S3 object. If you have a larger average item size, place proportionally fewer items into each S3 object.

Randomize sorted data

If an S3 object holds data in sorted order, it can create a *rolling hot partition*. This is a situation where one partition receives all the activity, and then the next partition after that, and so on. Data in sorted order is defined as items in sequence in the S3 object that will be written to the same target partition during the import. One common situation where data is in sorted order is a CSV file where items are sorted by partition key so that repeated items share the same partition key.

To avoid a rolling hot partition, we recommend that you randomize the order in these cases. This can improve performance by spreading the write operations. For more information, see Distributing write activity efficiently during data upload in DynamoDB.

Compress data to keep the total S3 object size below the Regional limit

In the import from S3 process, there is a limit on the sum total size of the S3 object data to be imported. The limit is 15 TB in the us-east-1, us-west-2, and eu-west-1 Regions, and 1 TB in all other Regions. The limit is based on the raw S3 object sizes.

Compression allows more raw data to fit within the limit. If compression alone isn't sufficient to fit the import within the limit, you can also contact AWS Premium Support for a quota increase.

Be aware of how item size impacts performance

If your average item size is very small (below 200 bytes), the import process might take a little longer than for larger item sizes.

Do not modify S3 objects during active imports

Ensure that your source S3 objects remain unchanged while an import operation is in progress. If an S3 object is modified during an import, the operation will fail with error code `ObjectModifiedInS3DuringImport` and the message "The S3 object could not be imported because it was overwritten."

If you encounter this error, restart the import operation with a stable version of your S3 object. To avoid this issue, wait for the current import to complete before making changes to the source files.

Consider importing without any Global Secondary Indexes

The duration of an import task may depend on the presence of one or multiple global secondary indexes (GSIs). If you plan to establish indexes with partition keys that have low cardinality, you may see a faster import if you defer index creation until after the import task is finished (rather than including them in the import job).

> **Note**
>
> Creating a GSI does not incur write charges, whether it is created during or after the import.

DynamoDB data export to Amazon S3: how it works

DynamoDB export to S3 is a fully managed solution for exporting your DynamoDB data to an Amazon S3 bucket at scale. Using DynamoDB export to S3, you can export data from an Amazon

DynamoDB table from any time within your point-in-time recovery (PITR) window to an Amazon S3 bucket. You need to enable PITR on your table to use the export functionality. This feature enables you to perform analytics and complex queries on your data using other AWS services such as Athena, AWS Glue, Amazon SageMaker AI, Amazon EMR, and AWS Lake Formation.

DynamoDB export to S3 allows you to export both full and incremental data from your DynamoDB table. Exports are asynchronous, they don't consume read capacity units (RCUs) and have no impact on table performance and availability. The export file formats supported are DynamoDB JSON and Amazon Ion formats. You can also export data to an S3 bucket owned by another AWS account and to a different AWS region. Your data is always encrypted end-to-end.

DynamoDB full exports are charged based on the size of the DynamoDB table (table data and local secondary indexes) at the point in time for which the export is done. DynamoDB incremental exports are charged based on the size of data processed from your continuous backups for the time period being exported. Incremental export has a minimum charge of 10MB. Additional charges apply for storing exported data in Amazon S3 and for PUT requests made against your Amazon S3 bucket. For more information about these charges, see Amazon DynamoDB pricing and Amazon S3 pricing.

For specifics on service quotas, see Table export to Amazon S3.

Topics

- Requesting a table export in DynamoDB
- DynamoDB table export output format

Requesting a table export in DynamoDB

DynamoDB table exports allow you to export table data to an Amazon S3 bucket, enabling you to perform analytics and complex queries on your data using other AWS services such as Athena, AWS Glue, Amazon SageMaker AI, Amazon EMR, and AWS Lake Formation. You can request a table export using the AWS Management Console, the AWS CLI, or the DynamoDB API.

> **Note**
>
> Requester pays Amazon S3 buckets aren't supported.

DynamoDB supports both full export and incremental export:

- With **full exports**, you can export a full snapshot of your table from any point in time within the point-in-time recovery (PITR) window to your Amazon S3 bucket.
- With **incremental exports**, you can export data from your DynamoDB table that was changed, updated, or deleted between a specified time period, within your PITR window, to your Amazon S3 bucket.

Topics

- Prerequisites
- Requesting an export using the AWS Management Console
- Getting details about past exports in the AWS Management Console
- Requesting an export using the AWS CLI and AWS SDKs
- Getting details about past exports using the AWS CLI and AWS SDKs

Prerequisites

Enable PITR

To use the export to S3 feature, you must enable PITR on your table. For details about how to enable PITR, see Point-in-time recovery. If you request an export for a table that doesn't have PITR enabled, your request will fail with an exception message: “An error occurred (PointInTimeRecoveryUnavailableException) when calling the `ExportTableToPointInTime` operation: Point in time recovery is not enabled for table 'my-dynamodb-table”. You can only request and export from a point in time that is within your configured PITR `RecoveryPeriodInDays`.

Set up S3 permissions

You can export your table data to any Amazon S3 bucket you have permission to write to. The destination bucket doesn't need to be in the same AWS Region or have the same owner as the source table owner. Your AWS Identity and Access Management (IAM) policy needs to allow you to be able to perform S3 actions (`s3:AbortMultipartUpload`, `s3:PutObject`, and `s3:PutObjectAcl`) and the DynamoDB export action (`dynamodb:ExportTableToPointInTime`). Here's an example of a sample policy that will grant your user permissions to perform exports to an S3 bucket.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "AllowDynamoDBExportAction",
            "Effect": "Allow",
            "Action": "dynamodb:ExportTableToPointInTime",
            "Resource": "arn:aws:dynamodb:us-east-1:111122223333:table/my-table"
        },
        {
            "Sid": "AllowS3BucketWrites",
            "Effect": "Allow",
            "Action": [
                "s3:AbortMultipartUpload",
                "s3:PutObject",
                "s3:PutObjectAcl"
            ],
            "Resource": "arn:aws:s3:::amzn-s3-demo-bucket/*"
        }
    ]
}
```

If you need to write to an Amazon S3 bucket that is in another account or you don't have permissions to write to, the Amazon S3 bucket owner must add a bucket policy to allow you to export from DynamoDB to that bucket. Here's an example policy on the target Amazon S3 bucket.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "ExampleStatement",
            "Effect": "Allow",
            "Principal": {
                "AWS": "arn:aws:iam::123456789012:user/Dave"
            },
            "Action": [
```

```
                "s3:AbortMultipartUpload",
                "s3:PutObject",
                "s3:PutObjectAcl"
            ],
            "Resource": "arn:aws:s3:::amzn-s3-demo-bucket/*"
        }
    ]
}
```

Revoking these permissions while an export is in progress will result in partial files.

> **Note**
>
> If the table or bucket you're exporting to is encrypted with customer managed keys, that KMS key's policies must give DynamoDB permission to use it. This permission is given through the IAM User/Role that triggers the export job. For more information on encryption including best practices, see How DynamoDB uses AWS KMS and Using a custom KMS key.

Requesting an export using the AWS Management Console

The following example demonstrates how to use the DynamoDB console to export an existing table named `MusicCollection`.

> **Note**
>
> This procedure assumes that you have enabled point-in-time recovery. To enable it for the `MusicCollection` table, on the table's **Overview** tab, in the **Table details** section, choose **Enable** for **Point-in-time recovery**.

To request a table export

1. Sign in to the AWS Management Console and open the DynamoDB console at https://console.aws.amazon.com/dynamodb/.
2. In the navigation pane on the left side of the console, choose **Exports to S3**.
3. Select the **Export to S3** button.

4. Choose a source table and destination S3 bucket. If the destination bucket is owned by your account, you can use the **Browse S3** button to find it. Otherwise, enter the URL of the bucket using the `s3://bucketname/prefix format.` the **`prefix`** is an optional folder to help keep your destination bucket organized.
5. Choose **Full export** or **Incremental export**. A **full export** outputs the full table snapshot of your table as it was at the point in time you specify. An **incremental export** outputs the changes made to your table during the specified export period. Your output is compacted so it only contains the final state of the item from the export period. The item will only appear once in the export even if it has multiple updates within the same export period.

   Full export

   1. Select the point in time you want to export the full table snapshot from. This can be any point in time within the PITR window. Alternatively, you can select **Current time** to export the latest snapshot.
   2. For **Exported file format**, choose between **DynamoDB JSON** and **Amazon Ion**. By default, your table will be exported in DynamoDB JSON format from the latest restorable time in the point in time recovery window and encrypted using an Amazon S3 key (SSE-S3). You can change these export settings if necessary.

      > **Note**
      >
      > If you choose to encrypt your export using a key protected by AWS Key Management Service (AWS KMS), the key must be in the same Region as the destination S3 bucket.

   Incremental export

   1. Select the **Export period** you want to export the incremental data for. Pick a start time within the PITR window. The export period duration must be at least 15 minutes and be no longer than 24 hours. The export period's start time is inclusive and the end time is exclusive.
   2. Choose between **Absolute mode** or **Relative mode**.

      a. **Absolute mode** will export incremental data for the time period you specify.

b. **Relative mode** will export incremental data for an export period that is relative to your export job submission time.

3. For **Exported file format**, choose between **DynamoDB JSON** and **Amazon Ion**. By default, your table will be exported in DynamoDB JSON format from the latest restorable time in the point in time recovery window and encrypted using an Amazon S3 key (SSE-S3). You can change these export settings if necessary.

> **Note**
>
> If you choose to encrypt your export using a key protected by AWS Key Management Service (AWS KMS), the key must be in the same Region as the destination S3 bucket.

4. For **Export view type**, select either **New and old images** or **New images only**. New image provides the latest state of the item. Old image provides the state of the item right before the specified “start date and time”. The default setting is **New and old images**. For more information on new images and old images, see Incremental export output.

6. Choose **Export** to begin.

Exported data isn't transactionally consistent. Your transaction operations can be torn between two export outputs. A subset of items can be modified by a transaction operation reflected in the export, while another subset of modifications in the same transaction isn't reflected in the same export request. However, exports are eventually consistent. If a transaction is torn during an export, you'll have the remaining transaction in your next contiguous export, without duplicates. The time periods used for exports are based on an internal system clock and can vary by one minute of your application’s local clock.

Getting details about past exports in the AWS Management Console

You can find information about export tasks you've run in the past by choosing the **Exports to S3** section in the navigation sidebar. This section contains a list of all exports you've created in the past 90 days. Select the ARN of a task listed in the **Exports** tab to retrieve information about that export, including any advanced configuration settings you chose. Note that although export task metadata expires after 90 days and jobs older than that are no longer found in this list, the objects in your S3 bucket remain as long as their bucket policies allow. DynamoDB never deletes any of the objects it creates in your S3 bucket during an export.

Requesting an export using the AWS CLI and AWS SDKs

The following examples show how to export an existing table to an S3 bucket.

> **Note**
>
> This procedure assumes that you have enabled point-in-time recovery. To enable it for the `MusicCollection` table, run the following command.
>
> ```
> aws dynamodb update-continuous-backups \
>     --table-name MusicCollection \
>     --point-in-time-recovery-specification PointInTimeRecoveryEnabled=True
> ```

Full export

AWS CLI

> **Note**
>
> If requesting a cross-account table export, make sure to include the `--s3-bucket-owner` option.

```
aws dynamodb export-table-to-point-in-time \
  --table-arn arn:aws:dynamodb:us-west-2:111122223333:table/MusicCollection \
  --s3-bucket ddb-export-musiccollection-9012345678 \
  --s3-prefix 2020-Nov \
  --export-format DYNAMODB_JSON \
  --export-time 1604632434 \
  --s3-bucket-owner 9012345678 \
  --s3-sse-algorithm AES256
```

Python

```
import boto3
from datetime import datetime

client = boto3.client('dynamodb')

client.export_table_to_point_in_time(
```

```
    TableArn='arn:aws:dynamodb:us-east-1:111122223333:table/TABLE',
    ExportTime=datetime(2023, 9, 20, 12, 0, 0),
    S3Bucket='bucket',
    S3Prefix='prefix',
    S3SseAlgorithm='AES256',
    ExportFormat='DYNAMODB_JSON'
)
```

Java

```
DynamoDbClient client = DynamoDbClient.create();

client.exportTableToPointInTime(b -> b
    .tableArn("arn:aws:dynamodb:us-east-1:111122223333:table/TABLE")
    .exportTime(Instant.parse("2023-09-20T12:00:00Z"))
    .s3Bucket("bucket")
    .s3Prefix("prefix")
    .s3SseAlgorithm(S3SseAlgorithm.AES256)
    .exportFormat(ExportFormat.DYNAMODB_JSON));
```

.NET

```
var client = new AmazonDynamoDBClient();

await client.ExportTableToPointInTimeAsync(new ExportTableToPointInTimeRequest
{
    TableArn = "arn:aws:dynamodb:us-east-1:111122223333:table/TABLE",
    ExportTime = new DateTime(2023, 9, 20, 12, 0, 0, DateTimeKind.Utc),
    S3Bucket = "bucket",
    S3Prefix = "prefix",
    S3SseAlgorithm = S3SseAlgorithm.AES256,
    ExportFormat = ExportFormat.DYNAMODB_JSON
});
```

JavaScript

```
import { DynamoDBClient, ExportTableToPointInTimeCommand } from "@aws-sdk/client-
dynamodb";

const client = new DynamoDBClient();

await client.send(new ExportTableToPointInTimeCommand({
    TableArn: "arn:aws:dynamodb:us-east-1:111122223333:table/TABLE",
```

```
    ExportTime: new Date("2023-09-20T12:00:00Z"),
    S3Bucket: "bucket",
    S3Prefix: "prefix",
    S3SseAlgorithm: "AES256",
    ExportFormat: "DYNAMODB_JSON"
}));
```

Go

```
cfg, _ := config.LoadDefaultConfig(context.TODO())
client := dynamodb.NewFromConfig(cfg)

exportTime := time.Date(2023, 9, 20, 12, 0, 0, 0, time.UTC)
client.ExportTableToPointInTime(context.TODO(),
 &dynamodb.ExportTableToPointInTimeInput{
    TableArn:       aws.String("arn:aws:dynamodb:us-east-1:111122223333:table/
TABLE"),
    ExportTime:     &exportTime,
    S3Bucket:       aws.String("bucket"),
    S3Prefix:       aws.String("prefix"),
    S3SseAlgorithm: types.S3SseAlgorithmAes256,
    ExportFormat:   types.ExportFormatDynamodbJson,
})
```

Incremental export

AWS CLI

```
aws dynamodb export-table-to-point-in-time \
  --table-arn arn:aws:dynamodb:REGION:ACCOUNT:table/TABLENAME \
  --s3-bucket BUCKET --s3-prefix PREFIX \
  --incremental-export-specification
 ExportFromTime=1693569600,ExportToTime=1693656000,ExportViewType=NEW_AND_OLD_IMAGES
 \
  --export-type INCREMENTAL_EXPORT
```

Python

```
import boto3
from datetime import datetime

client = boto3.client('dynamodb')
```

```
client.export_table_to_point_in_time(
    TableArn='arn:aws:dynamodb:us-east-1:111122223333:table/TABLE',
    IncrementalExportSpecification={
      'ExportFromTime': datetime(2023, 9, 20, 12, 0, 0),
      'ExportToTime': datetime(2023, 9, 20, 13, 0, 0),
      'ExportViewType': 'NEW_AND_OLD_IMAGES'
    },
    ExportType='INCREMENTAL_EXPORT',
    S3Bucket='bucket',
    S3Prefix='prefix',
    S3SseAlgorithm='AES256',
    ExportFormat='DYNAMODB_JSON'
)
```

Java

```
DynamoDbClient client = DynamoDbClient.create();

client.exportTableToPointInTime(b -> b
    .tableArn("arn:aws:dynamodb:us-east-1:111122223333:table/TABLE")
    .exportType(ExportType.INCREMENTAL_EXPORT)
    .incrementalExportSpecification(i -> i
        .exportFromTime(Instant.parse("2023-09-20T12:00:00Z"))
        .exportToTime(Instant.parse("2023-09-20T13:00:00Z"))
        .exportViewType(ExportViewType.NEW_AND_OLD_IMAGES))
    .s3Bucket("bucket")
    .s3Prefix("prefix")
    .s3SseAlgorithm(S3SseAlgorithm.AES256)
    .exportFormat(ExportFormat.DYNAMODB_JSON));
```

.NET

```
var client = new AmazonDynamoDBClient();

await client.ExportTableToPointInTimeAsync(new ExportTableToPointInTimeRequest
{
    TableArn = "arn:aws:dynamodb:us-east-1:111122223333:table/TABLE",
    ExportType = ExportType.INCREMENTAL_EXPORT,
    IncrementalExportSpecification = new IncrementalExportSpecification
    {
        ExportFromTime = new DateTime(2023, 9, 20, 12, 0, 0, DateTimeKind.Utc),
        ExportToTime = new DateTime(2023, 9, 20, 13, 0, 0, DateTimeKind.Utc),
```

```
        ExportViewType = ExportViewType.NEW_AND_OLD_IMAGES
    },
    S3Bucket = "bucket",
    S3Prefix = "prefix",
    S3SseAlgorithm = S3SseAlgorithm.AES256,
    ExportFormat = ExportFormat.DYNAMODB_JSON
});
```

JavaScript

```
import { DynamoDBClient, ExportTableToPointInTimeCommand } from "@aws-sdk/client-
dynamodb";

const client = new DynamoDBClient();

await client.send(new ExportTableToPointInTimeCommand({
    TableArn: "arn:aws:dynamodb:us-east-1:111122223333:table/TABLE",
    ExportType: "INCREMENTAL_EXPORT",
    IncrementalExportSpecification: {
        ExportFromTime: new Date("2023-09-20T12:00:00Z"),
        ExportToTime: new Date("2023-09-20T13:00:00Z"),
        ExportViewType: "NEW_AND_OLD_IMAGES"
    },
    S3Bucket: "bucket",
    S3Prefix: "prefix",
    S3SseAlgorithm: "AES256",
    ExportFormat: "DYNAMODB_JSON"
}));
```

Go

```
cfg, _ := config.LoadDefaultConfig(context.TODO())
client := dynamodb.NewFromConfig(cfg)

fromTime := time.Date(2023, 9, 20, 12, 0, 0, 0, time.UTC)
toTime := time.Date(2023, 9, 20, 13, 0, 0, 0, time.UTC)
client.ExportTableToPointInTime(context.TODO(),
 &dynamodb.ExportTableToPointInTimeInput{
    TableArn:   aws.String("arn:aws:dynamodb:us-east-1:111122223333:table/TABLE"),
    ExportType: types.ExportTypeIncrementalExport,
    IncrementalExportSpecification: &types.IncrementalExportSpecification{
        ExportFromTime: &fromTime,
        ExportToTime:   &toTime,
```

```
        ExportViewType: types.ExportViewTypeNewAndOldImages,
    },
    S3Bucket:       aws.String("bucket"),
    S3Prefix:       aws.String("prefix"),
    S3SseAlgorithm: types.S3SseAlgorithmAes256,
    ExportFormat:   types.ExportFormatDynamodbJson,
})
```

> **Note**
>
> If you choose to encrypt your export using a key protected by AWS Key Management Service (AWS KMS), the key must be in the same Region as the destination S3 bucket.

Getting details about past exports using the AWS CLI and AWS SDKs

You can find information about export requests you've run in the past by using the `list-exports` command. This command returns a list of all exports you've created in the past 90 days. Note that although export task metadata expires after 90 days and jobs older than that are no longer returned by the `list-exports` command, the objects in your S3 bucket remain as long as their bucket policies allow. DynamoDB never deletes any of the objects it creates in your S3 bucket during an export.

Exports have a status of `PENDING` until they either succeed or fail. If they succeed, the status changes to `COMPLETED`. If they fail, the status changes to `FAILED` with a `failure_message` and `failure_reason`.

List exports

AWS CLI

```
aws dynamodb list-exports \
    --table-arn arn:aws:dynamodb:us-east-1:111122223333:table/ProductCatalog
```

Python

```
import boto3

client = boto3.client('dynamodb')
```

```
print(
  client.list_exports(
     TableArn='arn:aws:dynamodb:us-east-1:111122223333:table/TABLE',
  )
)
```

Java

```
DynamoDbClient client = DynamoDbClient.create();

ListExportsResponse response = client.listExports(b -> b
    .tableArn("arn:aws:dynamodb:us-east-1:111122223333:table/TABLE"));

response.exportSummaries().forEach(System.out::println);
```

.NET

```
var client = new AmazonDynamoDBClient();

var response = await client.ListExportsAsync(new ListExportsRequest
{
    TableArn = "arn:aws:dynamodb:us-east-1:111122223333:table/TABLE"
});

response.ExportSummaries.ForEach(Console.WriteLine);
```

JavaScript

```
import { DynamoDBClient, ListExportsCommand } from "@aws-sdk/client-dynamodb";

const client = new DynamoDBClient();

const response = await client.send(new ListExportsCommand({
    TableArn: "arn:aws:dynamodb:us-east-1:111122223333:table/TABLE"
}));

console.log(response.ExportSummaries);
```

Go

```
cfg, _ := config.LoadDefaultConfig(context.TODO())
client := dynamodb.NewFromConfig(cfg)
```

```
response, _ := client.ListExports(context.TODO(), &dynamodb.ListExportsInput{
    TableArn: aws.String("arn:aws:dynamodb:us-east-1:111122223333:table/TABLE"),
})

fmt.Println(response.ExportSummaries)
```

Describe export

AWS CLI

```
aws dynamodb describe-export \
    --export-arn arn:aws:dynamodb:us-east-1:111122223333:table/ProductCatalog/
export/01695353076000-a1b2c3d4
```

Python

```
import boto3

client = boto3.client('dynamodb')

print(
  client.describe_export(
     ExportArn='arn:aws:dynamodb:us-east-1:111122223333:table/TABLE/
export/01695353076000-06e2188f',
  )['ExportDescription']
)
```

Java

```
DynamoDbClient client = DynamoDbClient.create();

DescribeExportResponse response = client.describeExport(b -> b
    .exportArn("arn:aws:dynamodb:us-east-1:111122223333:table/TABLE/
export/01695353076000-06e2188f"));

System.out.println(response.exportDescription());
```

.NET

```
var client = new AmazonDynamoDBClient();
```

```
var response = await client.DescribeExportAsync(new DescribeExportRequest
{
    ExportArn = "arn:aws:dynamodb:us-east-1:111122223333:table/TABLE/
export/01695353076000-06e2188f"
});

Console.WriteLine(response.ExportDescription);
```

JavaScript

```
import { DynamoDBClient, DescribeExportCommand } from "@aws-sdk/client-dynamodb";

const client = new DynamoDBClient();

const response = await client.send(new DescribeExportCommand({
    ExportArn: "arn:aws:dynamodb:us-east-1:111122223333:table/TABLE/
export/01695353076000-06e2188f"
}));

console.log(response.ExportDescription);
```

Go

```
cfg, _ := config.LoadDefaultConfig(context.TODO())
client := dynamodb.NewFromConfig(cfg)

response, _ := client.DescribeExport(context.TODO(), &dynamodb.DescribeExportInput{
    ExportArn: aws.String("arn:aws:dynamodb:us-east-1:111122223333:table/TABLE/
export/01695353076000-06e2188f"),
})

fmt.Println(response.ExportDescription)
```

DynamoDB table export output format

A DynamoDB table export includes manifest files in addition to the files containing your table data. These files are all saved in the Amazon S3 bucket that you specify in your export request. The following sections describe the format and contents of each output object.

Full export output

Manifest files

DynamoDB creates manifest files, along with their checksum files, in the specified S3 bucket for each export request.

```
export-prefix/AWSDynamoDB/ExportId/manifest-summary.json
export-prefix/AWSDynamoDB/ExportId/manifest-summary.checksum
export-prefix/AWSDynamoDB/ExportId/manifest-files.json
export-prefix/AWSDynamoDB/ExportId/manifest-files.checksum
```

You choose an **`export-prefix`** when you request a table export. This helps you keep files in the destination S3 bucket organized. The **`ExportId`** is a unique token generated by the service to ensure that multiple exports to the same S3 bucket and `export-prefix` don't overwrite each other.

The export creates at least 1 file per partition. For partitions that are empty, your export request will create an empty file. All of the items in each file are from that particular partition's hashed keyspace.

> **Note**
>
> DynamoDB also creates an empty file named `_started` in the same directory as the manifest files. This file verifies that the destination bucket is writable and that the export has begun. It can safely be deleted.

The summary manifest

The `manifest-summary.json` file contains summary information about the export job. This allows you to know which data files in the shared data folder are associated with this export. Its format is as follows:

```
{
   "version": "2020-06-30",
   "exportArn": "arn:aws:dynamodb:us-east-1:123456789012:table/ProductCatalog/
export/01234567890123-a1b2c3d4",
   "startTime": "2020-11-04T07:28:34.028Z",
   "endTime": "2020-11-04T07:33:43.897Z",
```

```
   "tableArn": "arn:aws:dynamodb:us-east-1:123456789012:table/ProductCatalog",
   "tableId": "12345a12-abcd-123a-ab12-1234abc12345",
   "exportTime": "2020-11-04T07:28:34.028Z",
   "s3Bucket": "ddb-productcatalog-export",
   "s3Prefix": "2020-Nov",
   "s3SseAlgorithm": "AES256",
   "s3SseKmsKeyId": null,
   "manifestFilesS3Key": "AWSDynamoDB/01693685827463-2d8752fd/manifest-files.json",
   "billedSizeBytes": 0,
   "itemCount": 8,
   "outputFormat": "DYNAMODB_JSON",
   "exportType": "FULL_EXPORT"
}
```

The files manifest

The `manifest-files.json` file contains information about the files that contain your exported table data. The file is in JSON lines format, so newlines are used as item delimiters. In the following example, the details of one data file from a files manifest are formatted on multiple lines for the sake of readability.

```
{
"itemCount": 8,
   "md5Checksum": "sQMSpEILNgoQmarvDFonGQ==",
   "etag": "af83d6f217c19b8b0fff8023d8ca4716-1",
   "dataFileS3Key": "AWSDynamoDB/01693685827463-2d8752fd/data/asdl123dasas.json.gz"
}
```

Data files

DynamoDB can export your table data in two formats: DynamoDB JSON and Amazon Ion. Regardless of the format you choose, your data will be written to multiple compressed files named by the keys. These files are also listed in the `manifest-files.json` file.

The directory structure of your Amazon S3 bucket after a full export will contain all of your manifest files and data files under the export Id folder.

```
amzn-s3-demo-bucket/DestinationPrefix
.
### AWSDynamoDB
    ### 01693685827463-2d8752fd     // the single full export
    #   ### manifest-files.json     // manifest points to files under 'data' subfolder
```

```
    #   ### manifest-files.checksum
    #   ### manifest-summary.json   // stores metadata about request
    #   ### manifest-summary.md5
    #   ### data                    // The data exported by full export
    #   #   ### asdl123dasas.json.gz
    #   #   ...
    #   ### _started                // empty file for permission check
```

DynamoDB JSON

A table export in DynamoDB JSON format consists of multiple `Item` objects. Each individual object is in DynamoDB's standard marshalled JSON format.

When creating custom parsers for DynamoDB JSON export data, the format is JSON lines. This means that newlines are used as item delimiters. Many AWS services, such as Athena and AWS Glue, will parse this format automatically.

In the following example, a single item from a DynamoDB JSON export has been formatted on multiple lines for the sake of readability.

```
{
    "Item":{
        "Authors":{
            "SS":[
                "Author1",
                "Author2"
            ]
        },
        "Dimensions":{
            "S":"8.5 x 11.0 x 1.5"
        },
        "ISBN":{
            "S":"333-3333333333"
        },
        "Id":{
            "N":"103"
        },
        "InPublication":{
            "BOOL":false
        },
        "PageCount":{
            "N":"600"
        },
```

```
        "Price":{
            "N":"2000"
        },
        "ProductCategory":{
            "S":"Book"
        },
        "Title":{
            "S":"Book 103 Title"
        }
    }
}
```

Amazon Ion

Amazon Ion is a richly-typed, self-describing, hierarchical data serialization format built to address rapid development, decoupling, and efficiency challenges faced every day while engineering large-scale, service-oriented architectures. DynamoDB supports exporting table data in Ion's text format, which is a superset of JSON.

When you export a table to Ion format, the DynamoDB datatypes used in the table are mapped to Ion datatypes. DynamoDB sets use Ion type annotations to disambiguate the datatype used in the source table.

The following table lists the mapping of DynamoDB data types to ion data types:

DynamoDB data type	Ion representation
String (S)	string
Boolean (BOOL)	bool
Number (N)	decimal
Binary (B)	blob
Set (SS, NS, BS)	list (with type annotation $dynamodb_SS, $dynamodb_NS, or $dynamodb_BS)
List	list
Map	struct

Items in an Ion export are delimited by newlines. Each line begins with an Ion version marker, followed by an item in Ion format. In the following example, an item from an Ion export has been formatted on multiple lines for the sake of readability.

```
$ion_1_0 {
    Item:{
        Authors:$dynamodb_SS::["Author1","Author2"],
        Dimensions:"8.5 x 11.0 x 1.5",
        ISBN:"333-3333333333",
        Id:103.,
        InPublication:false,
        PageCount:6d2,
        Price:2d3,
        ProductCategory:"Book",
        Title:"Book 103 Title"
    }
}
```

Incremental export output

Manifest files

DynamoDB creates manifest files, along with their checksum files, in the specified S3 bucket for each export request.

```
export-prefix/AWSDynamoDB/ExportId/manifest-summary.json
export-prefix/AWSDynamoDB/ExportId/manifest-summary.checksum
export-prefix/AWSDynamoDB/ExportId/manifest-files.json
export-prefix/AWSDynamoDB/ExportId/manifest-files.checksum
```

You choose an **`export-prefix`** when you request a table export. This helps you keep files in the destination S3 bucket organized. The **`ExportId`** is a unique token generated by the service to ensure that multiple exports to the same S3 bucket and `export-prefix` don't overwrite each other.

The export creates at least 1 file per partition. For partitions that are empty, your export request will create an empty file. All of the items in each file are from that particular partition's hashed keyspace.

> **Note**
>
> DynamoDB also creates an empty file named `_started` in the same directory as the manifest files. This file verifies that the destination bucket is writable and that the export has begun. It can safely be deleted.

The summary manifest

The `manifest-summary.json` file contains summary information about the export job. This allows you to know which data files in the shared data folder are associated with this export. Its format is as follows:

```
{
 "version": "2023-08-01",
 "exportArn": "arn:aws:dynamodb:us-east-1:599882009758:table/export-test/
export/01695097218000-d6299cbd",
 "startTime": "2023-09-19T04:20:18.000Z",
 "endTime": "2023-09-19T04:40:24.780Z",
 "tableArn": "arn:aws:dynamodb:us-east-1:599882009758:table/export-test",
 "tableId": "b116b490-6460-4d4a-9a6b-5d360abf4fb3",
 "exportFromTime": "2023-09-18T17:00:00.000Z",
 "exportToTime": "2023-09-19T04:00:00.000Z",
 "s3Bucket": "jason-exports",
 "s3Prefix": "20230919-prefix",
 "s3SseAlgorithm": "AES256",
 "s3SseKmsKeyId": null,
 "manifestFilesS3Key": "20230919-prefix/AWSDynamoDB/01693685934212-ac809da5/manifest-
files.json",
 "billedSizeBytes": 20901239349,
 "itemCount": 169928274,
 "outputFormat": "DYNAMODB_JSON",
 "outputView": "NEW_AND_OLD_IMAGES",
 "exportType": "INCREMENTAL_EXPORT"
}
```

The files manifest

The `manifest-files.json` file contains information about the files that contain your exported table data. The file is in JSON lines format, so newlines are used as item delimiters. In the following

example, the details of one data file from a files manifest are formatted on multiple lines for the sake of readability.

```
{
"itemCount": 8,
   "md5Checksum": "sQMSpEILNgoQmarvDFonGQ==",
   "etag": "af83d6f217c19b8b0fff8023d8ca4716-1",
   "dataFileS3Key": "AWSDynamoDB/data/sgad6417s6vss4p7owp0471bcq.json.gz"
}
```

Data files

DynamoDB can export your table data in two formats: DynamoDB JSON and Amazon Ion. Regardless of the format you choose, your data will be written to multiple compressed files named by the keys. These files are also listed in the `manifest-files.json` file.

The data files for incremental exports are all contained in a common data folder in your S3 bucket. Your manifest files are under your export ID folder.

```
amzn-s3-demo-bucket/DestinationPrefix
.
### AWSDynamoDB
    ### 01693685934212-ac809da5     // an incremental export ID
    #   ### manifest-files.json     // manifest points to files under 'data' folder
    #   ### manifest-files.checksum
    #   ### manifest-summary.json   // stores metadata about request
    #   ### manifest-summary.md5
    #   ### _started                // empty file for permission check
    ### 01693686034521-ac809da5
    #   ### manifest-files.json
    #   ### manifest-files.checksum
    #   ### manifest-summary.json
    #   ### manifest-summary.md5
    #   ### _started
    ### data                       // stores all the data files for incremental
 exports
    #   ### sgad6417s6vss4p7owp0471bcq.json.gz
    #   ...
```

In you export files, each item's output includes a timestamp that represents when that item was updated in your table and a data structure that indicates if it was an `insert`, `update`, or `delete` operation. The timestamp is based on an internal system clock and can vary from your application

clock. For incremental exports, you can choose between two export view types for your output structure: **new and old images** or **new images only**.

- **New image** provides the latest state of the item
- **Old image** provides the state of the item right before the specified **start date and time**

View types can be helpful if you want to see how the item was changed within the export period. It can also be useful for efficiently updating your downstream systems, especially if those downstream systems have a partition key that is not the same as your DynamoDB partition key.

You can infer whether an item in your incremental export output was an `insert`, `update`, or `delete` by looking at the structure of the output. The incremental export structure and its corresponding operations are summarized in the table below for both export view types.

Operation	New images only	New and old images
Insert	Keys + new image	Keys + new image
Update	Keys + new image	Keys + new image + old image
Delete	Keys	Keys + old image
Insert + delete	No output	No output

DynamoDB JSON

A table export in DynamoDB JSON format consists of a metadata timestamp that indicates the write time of the item, followed by the keys of the item and the values. The following shows an example DynamoDB JSON output using export view type as **New and Old images**.

```
// Ex 1: Insert
//   An insert means the item did not exist before the incremental export window
//   and was added during the incremental export window

{
   "Metadata": {
     "WriteTimestampMicros": "1680109764000000"
   },
```

```
   "Keys": {
     "PK": {
       "S": "CUST#100"
     }
   },
   "NewImage": {
     "PK": {
       "S": "CUST#100"
     },
     "FirstName": {
       "S": "John"
     },
     "LastName": {
       "S": "Don"
     }
   }
}

// Ex 2: Update
//   An update means the item existed before the incremental export window
//   and was updated during the incremental export window.
//   The OldImage would not be present if choosing "New images only".

{
   "Metadata": {
     "WriteTimestampMicros": "1680109764000000"
   },
   "Keys": {
     "PK": {
       "S": "CUST#200"
     }
   },
   "OldImage": {
     "PK": {
       "S": "CUST#200"
     },
     "FirstName": {
       "S": "Mary"
     },
     "LastName": {
       "S": "Grace"
     }
   },
   "NewImage": {
```

```
     "PK": {
       "S": "CUST#200"
     },
     "FirstName": {
       "S": "Mary"
     },
     "LastName": {
       "S": "Smith"
     }
   }
}

// Ex 3: Delete
//   A delete means the item existed before the incremental export window
//   and was deleted during the incremental export window
//   The OldImage would not be present if choosing "New images only".

{
   "Metadata": {
     "WriteTimestampMicros": "1680109764000000"
   },
   "Keys": {
     "PK": {
       "S": "CUST#300"
     }
   },
   "OldImage": {
     "PK": {
       "S": "CUST#300"
     },
     "FirstName": {
       "S": "Jose"
     },
     "LastName": {
       "S": "Hernandez"
     }
   }
}

// Ex 4: Insert + Delete
//   Nothing is exported if an item is inserted and deleted within the
//   incremental export window.
```

Amazon Ion

Amazon Ion is a richly-typed, self-describing, hierarchical data serialization format built to address rapid development, decoupling, and efficiency challenges faced every day while engineering large-scale, service-oriented architectures. DynamoDB supports exporting table data in Ion's text format, which is a superset of JSON.

When you export a table to Ion format, the DynamoDB datatypes used in the table are mapped to Ion datatypes. DynamoDB sets use Ion type annotations to disambiguate the datatype used in the source table.

The following table lists the mapping of DynamoDB data types to ion data types:

DynamoDB data type	Ion representation
String (S)	string
Boolean (BOOL)	bool
Number (N)	decimal
Binary (B)	blob
Set (SS, NS, BS)	list (with type annotation $dynamodb_SS, $dynamodb_NS, or $dynamodb_BS)
List	list
Map	struct

Items in an Ion export are delimited by newlines. Each line begins with an Ion version marker, followed by an item in Ion format. In the following example, an item from an Ion export has been formatted on multiple lines for the sake of readability.

```
$ion_1_0 {
    Record:{
        Keys:{
             ISBN:"333-3333333333"
         },
        Metadata:{
```

```
            WriteTimestampMicros:1684374845117899.
        },
        OldImage:{
            Authors:$dynamodb_SS::["Author1","Author2"],
            ISBN:"333-3333333333",
            Id:103.,
            InPublication:false,
            ProductCategory:"Book",
            Title:"Book 103 Title"
        },
        NewImage:{
            Authors:$dynamodb_SS::["Author1","Author2"],
            Dimensions:"8.5 x 11.0 x 1.5",
            ISBN:"333-3333333333",
            Id:103.,
            InPublication:true,
            PageCount:6d2,
            Price:2d3,
            ProductCategory:"Book",
            Title:"Book 103 Title"
        }
    }
}
```

DynamoDB zero-ETL integration with Amazon SageMaker Lakehouse

DynamoDB zero-ETL integration with Amazon SageMaker Lakehouse eliminates the need to build custom data movement pipelines by automatically replicating DynamoDB data to Amazon SageMaker Lakehouse. This no-code integration helps customers run analytics workloads on their DynamoDB data using Amazon SageMaker Lakehouse without consuming any DynamoDB table capacity. The integration automatically exports data from your table and keeps the target fresh, typically within 15 to 30 minutes.

Topics

- DynamoDB zero-ETL integration with Amazon SageMaker Lakehouse

DynamoDB zero-ETL integration with Amazon SageMaker Lakehouse

Setting up an integration between the DynamoDB table and Amazon SageMaker Lakehouse require prerequisites such as configuring IAM roles which AWS Glue uses to access data from the source and write to the target, and the use of KMS keys to encrypt the data in intermediate or the target location.

Topics

- Prerequisites before creating a DynamoDB zero-ETL integration with Amazon SageMaker Lakehouse
- Creating a DynamoDB zero-ETL integration with Amazon SageMaker Lakehouse
- Viewing CloudWatch metrics for integration

Prerequisites before creating a DynamoDB zero-ETL integration with Amazon SageMaker Lakehouse

To configure a zero-ETL integration with an DynamoDB source, you need to set up a Resource-Based Access (RBAC) policy that allows AWS Glue to access and export data from the DynamoDB table. The policy should include specific permissions like `ExportTableToPointInTime`, `DescribeTable`, and `DescribeExport` with conditions restricting access to a specific AWS account and region. See, Configuring an Amazon DynamoDB source for more information.

Point-in-time recovery (PITR) must be enabled for the table, and you can apply the policy using AWS CLI commands. The policy can be further refined by specifying the full integration ARN for more restrictive access control. For more information, see Prerequisites for setting up a zero-ETL integration.

Creating a DynamoDB zero-ETL integration with Amazon SageMaker Lakehouse

After completing integration prerequisites, you can create, modify, or delete the zero-ETL integration following the guidance below:

Creating an integration

To create an integration

1. Sign in to the AWS Management Console and open the Amazon DynamoDB console at https://console.aws.amazon.com/dynamodbv2.

2. In the navigation pane, choose **Integrations**.
3. Select **Create zero-ETL integration with Amazon SageMaker Lakehouse**, and then choose **Next**.
4. To create an integration, see Creating an integration.
5. To modify an integration, see Modifying an integration.
6. To delete an integration, see Deleting an integration.
7. To set up a cross-account integration, see Setting up cross-account integration.

Enabling compaction on target Amazon S3 tables

You can enable compaction to improve query performance in Amazon Athena.

First, complete the prerequisite setup for compaction resources, including configuring the necessary IAM role. Refer to the Lake Formation documentation for detailed IAM role configuration steps. See, Optimizing tables for compaction.

To enable compaction on the AWS Glue table created during integration, follow the Lake Formation compaction enabling process. This will help optimize your table's performance and query efficiency.

Viewing CloudWatch metrics for integration

Once an integration completes, you can see these CloudWatch metrics and EventBridge notifications generated in your account for each AWS Glue job. For more information, see Monitoring an integration.

DynamoDB zero-ETL integration with Amazon OpenSearch Service

Amazon DynamoDB offers a zero-ETL integration with Amazon OpenSearch Service through the **DynamoDB plugin for OpenSearch Ingestion**. Amazon OpenSearch Ingestion offers a fully managed, no-code experience for ingesting data into Amazon OpenSearch Service.

With the DynamoDB plugin for OpenSearch Ingestion, you can use one or more DynamoDB tables as a source for ingestion to one or more OpenSearch Service indexes. You can browse and configure your OpenSearch Ingestion pipelines with DynamoDB as a source from either OpenSearch Ingestion or DynamoDB Integrations in the AWS Management Console.

- Get started with OpenSearch Ingestion by following along in the OpenSearch Ingestion getting started guide.
- Learn about the prerequisites and all the configuration options for the DynamoDB plugin at DynamoDB plugin for OpenSearch Ingestion documentation.

How it works

The plugin uses DynamoDB export to Amazon S3 to create an initial snapshot to load into OpenSearch. After the snapshot has been loaded, the plugin uses DynamoDB Streams to replicate any further changes in near real time. Every item is processed as an event in OpenSearch Ingestion and can be modified with processor plugins. You can drop attributes or create composite attributes and send them to different indexes through routes.

You must have point-in-time recovery (PITR) enabled to use export to Amazon S3. You must also have DynamoDB Streams enabled (with the **new & old images** option selected) to be able to use it. It's possible to create a pipeline without taking a snapshot by excluding export settings.

You can also create a pipeline with only a snapshot and no updates by excluding streams settings. The plugin does not use read or write throughput on your table, so it is safe to use without impacting your production traffic. There are limits to the number of parallel consumers on a stream that you should consider before creating this or other integrations. For other considerations, see the section called “Integration best practices”.

For simple pipelines, a single OpenSearch Compute Unit (OCU) can process about 1 MB per second of writes. This is the equivalent of about 1000 write request units (WCU). Depending on your pipeline's complexity and other factors, you might achieve more or less than this.

OpenSearch Ingestion supports a dead-letter queue (DLQ) for events that cause unrecoverable errors. Additionally, the pipeline can resume from where it left off without user intervention even if there's an interruption of service with either DynamoDB, the pipeline, or Amazon OpenSearch Service.

If interruption goes on for longer than 24 hours, this can cause a loss of updates. However, the pipeline would continue to process the updates that were still available when availability is restored. You would need to do a fresh index build to fix any irregularities due to the dropped events unless they were in the dead-letter queue.

For all the settings and details for the plugin, see OpenSearch Ingestion DynamoDB plugin documentation.

Integrated create experience through the console

DynamoDB and OpenSearch Service have an integrated experience in the AWS Management Console, which streamlines the getting started process. When you go through these steps, the service will automatically select the DynamoDB blueprint and add the appropriate DynamoDB information for you.

To create an integration, follow along in the OpenSearch Ingestion getting started guide. When you get to Step 3: Create a pipeline, replace Steps 1 and 2 with the following steps:

1. Navigate to the DynamoDB console.
2. In the left-hand navigation pane, choose **Integration**.
3. Select the DynamoDB table that you'd like to replicate to OpenSearch.
4. Choose **Create**.

From here, you can continue on with the rest of the tutorial.

Next steps

For a better understanding of how DynamoDB integrates with OpenSearch Service, see the following:

- Getting started with Amazon OpenSearch Ingestion
- DynamoDB plugin configuration and requirements

Handling breaking changes to your index

OpenSearch can dynamically add new attributes to your index. However, after your mapping template has been set for a given key, you'll need to take additional action to change it. Additionally, if your change requires you to reprocess all the data in your DynamoDB table, you'll need to take steps to initiate a fresh export.

> **Note**
>
> In all these options, you might still run into issues if your DynamoDB table has type conflicts with the mapping template you've specified. Ensure that you have a dead-letter queue (DLQ) enabled (even in development). This makes it easier to understand what might

be wrong with the record that causes a conflict when it's being indexed into your index on OpenSearch.

Topics

- How it works
- Delete your index and reset the pipeline (pipeline-centric option)
- Recreate your index and reset the pipeline (index-centric option)
- Create a new index and sink (online option)
- Best practices for avoiding and debugging type conflicts

How it works

Here's a quick overview of the actions taken when handling breaking changes to your index. See the step-by-step procedures in the sections that follow.

- **Stop and start the pipeline**: This option resets the pipeline's state, and the pipeline will restart with a new full export. It is non-destructive, so it does **not** delete your index or any data in DynamoDB. If you don't create a fresh index before you do this, you might see a high number of errors from version conflicts because the export tries to insert older documents than the current `_version` in the index. You can safely ignore these errors. You will not be billed for the pipeline while it is stopped.
- **Update the pipeline**: This option updates the configuration in the pipeline with a blue/green approach, without losing any state. If you make significant changes to your pipeline (such as adding new routes, indexes, or keys to existing indexes), you might need to do a full reset of the pipeline and recreate your index. This option does **not** perform a full export.
- **Delete and recreate the index**: This option removes your data and mapping settings on your index. You should do this before making any breaking changes to your mappings. It will break any applications that rely on the index until the index is recreated and synchronized. Deleting the index does **not** initiate a fresh export. You should delete your index only after you've updated your pipeline. Otherwise, your index might be recreated before you update your settings.

Delete your index and reset the pipeline (pipeline-centric option)

This method is often the fastest option if you're still in development. You'll delete your index in OpenSearch Service, and then stop and start your pipeline to initiate a fresh export of all your data. This ensures that there are no mapping template conflicts with existing indexes, and no loss of data from an incomplete processed table.

1. Stop the pipeline either through the AWS Management Console, or by using the StopPipeline API operation with the AWS CLI or an SDK.
2. Update your pipeline configuration with your new changes.
3. Delete your index in OpenSearch Service, either through a REST API call or your OpenSearch Dashboard.
4. Start the pipeline either through the console, or by using the `StartPipeline` API operation with the AWS CLI or an SDK.

   > **Note**
   >
   > This initiates a fresh full export, which will incur additional costs.

5. Monitor for any unexpected issues because a fresh export is generated to create the new index.
6. Confirm that the index matches your expectations in OpenSearch Service.

After the export has completed and it resumes reading from the stream, your DynamoDB table data will now be available in the index.

Recreate your index and reset the pipeline (index-centric option)

This method works well if you need to do a lot of iterations on the index design in OpenSearch Service before resuming the pipeline from DynamoDB. This can be useful for development when you want to iterate very quickly on your search patterns, and want to avoid waiting on fresh exports to complete between each iteration.

1. Stop the pipeline either through the AWS Management Console, or by calling the StopPipeline API operation with the AWS CLI or an SDK.
2. Delete and recreate your index in OpenSearch with the mapping template you want to use. You can manually insert some sample data to confirm that your searches are working as

intended. If your sample data might conflict with any data from DynamoDB, be sure to delete it before moving onto the next step.

3. If you have an indexing template in your pipeline, remove it or replace it with the one you've created already in OpenSearch Service. Ensure that the name of your index matches the name in the pipeline.
4. Start the pipeline either through console, or by calling the `StartPipeline` API operation with the AWS CLI or an SDK.

   > **Note**
   >
   > This will initiate a fresh full export, which will incur additional costs.

5. Monitor for any unexpected issues because a fresh export is generated to create the new index.

After the export has completed and it resumes reading from the stream, you should be your DynamoDB table data will now be available in the index.

Create a new index and sink (online option)

This method works well if you need to update your mapping template but are currently using your index in production. This creates a brand new index, which you'll need to move your application over to after it's synchronized and validated.

> **Note**
>
> This will create another consumer on the stream. This can be an issue if you also have other consumers like AWS Lambda or global tables. You might need to pause updates to your existing pipeline to create capacity to load the new index.

1. Create a new pipeline with new settings and a different index name.
2. Monitor the new index for any unexpected issues.
3. Swap the application over to the new index.
4. Stop and delete the old pipeline after validating that everything is working correctly.

Best practices for avoiding and debugging type conflicts

- Always use a dead-letter queue (DLQ) to make it easier to debug when there are type conflicts.
- Always use an index template with mappings and set `include_keys`. While OpenSearch Service dynamically maps new keys, this can cause issues with unexpected behaviors (such as expecting something to be a `GeoPoint`, but it's created as a `string` or `object`) or errors (such as having a `number` that is a mix of `long` and `float` values).
- If you need to keep your existing index working in production, you can also replace any of the previous delete index steps with just renaming your index in your pipeline config file. This creates a brand new index. Your application will then need to be updated to point to the new index after it's complete.
- If you have a type conversion issue that you fix with a processor, you can test this with `UpdatePipeline`. To do this, you'll need to do a stop and start or process your dead-letter queues to fix any previously skipped documents that had errors.

Best practices for working with DynamoDB zero-ETL integration and OpenSearch Service

DynamoDB has a DynamoDB zero-ETL integration with Amazon OpenSearch Service. For more information, see the DynamoDB plugin for OpenSearch Ingestion and specific best practices for Amazon OpenSearch Service.

Configuration

- Only index data that you need to perform searches on. Always use a mapping template (`template_type: index_template` and `template_content`) and `include_keys` to implement this.
- Monitor your logs for errors that are related to type conflicts. OpenSearch Service expects all values for a given key to have the same type. It generates exceptions if there's a mismatch. If you encounter one of these errors, you can add a processor to catch that a given key is always be the same value.
- Generally use the `primary_key` metadata value for the `document_id` value. In OpenSearch Service, the document ID is the equivalent of the primary key in DynamoDB. Using the primary key will make it easy to find your document and ensure that updates are consistently replicated to it without conflicts.

You can use the helper function `getMetadata` to get your primary key (for example, `document_id: "${getMetadata('primary_key')}"`). If you're using a composite primary key, the helper function will concatenate them together for you.

- In general, use the `opensearch_action` metadata value for the `action` setting. This will ensure that updates are replicated in such a way that the data in OpenSearch Service matches the latest state in DynamoDB.

  You can use the helper function `getMetadata` to get your primary key (for example, `action: "${getMetadata('opensearch_action')}"`). You can also get the stream event type through `dynamodb_event_name` for use cases like filtering. However, you should typically not use it for the `action` setting.

Observability

- Always use a dead-letter queue (DLQ) on your OpenSearch sinks to handle dropped events. DynamoDB is generally less structured than OpenSearch Service, and it's always possible for something unexpected to happen. With a dead-letter queue, you can recover individual events, and even automate the recovery process. This will help you to avoid needing to rebuild your entire index.
- Always set alerts that your replication delay doesn't go over an expected amount. It is typically safe to assume one minute without the alert being too noisy. This can vary depending on how spiky your write traffic is and your OpenSearch Compute Unit (OCU) settings on the pipeline.

  If your replication delay goes over 24 hours, your stream will start to drop events, and you'll have accuracy issues unless you do a full rebuild of your index from scratch.

Scaling

- Use auto scaling for pipelines to help scale up or down the OCUs to best fit the workload.
- For provisioned throughput tables without auto scaling, we recommend setting OCUs based on your write capacity units (WCUs) divided by 1000. Set the minimum to 1 OCU below that amount (but at least 1), and set the maximum to at least 1 OCU above that amount.
  - **Formula:**

```
OCU_minimum = GREATEST((table_WCU / 1000) - 1, 1)
```

```
OCU_maximum = (table_WCU / 1000) + 1
```

- **Example:** Your table has 25000 WCUs provisioned. Your pipeline's OCUs should be set with a minimum of 24 (25000/1000 - 1) and maximum of at least 26 (25000/1000 + 1).

- For provisioned throughput tables with auto scaling, we recommend setting OCUs based on your minimum and maximum WCUs, divided by 1000. Set the minimum to 1 OCU below the minimum from DynamoDB, and set the maximum to at least 1 OCU above the maximum from DynamoDB.
  - **Formula:**

```
OCU_minimum = GREATEST((table_minimum_WCU / 1000) - 1, 1)
OCU_maximum = (table_maximum_WCU / 1000) + 1
```

  - **Example:** Your table has an auto scaling policy with a minimum of 8000 and maximum of 14000. Your pipeline's OCUs should be set with a minimum of 7 (8000/1000 - 1) and a maximum of 15 (14000/1000 + 1).
- For on-demand throughput tables, we recommend setting OCUs based on your typical peak and valley for write request units per second. You might need to average over a longer time period, depending on the aggregation that's available to you. Set the minimum to 1 OCU below the minimum from DynamoDB, and set the maximum to at least 1 OCU above the maximum from DynamoDB.
  - **Formula:**

```
# Assuming we have writes aggregated at the minute level
OCU_minimum = GREATEST((min(table_writes_1min) / (60 * 1000)) - 1, 1)
OCU_maximum = (max(table_writes_1min) / (60 * 1000)) + 1
```

  - **Example:** Your table has an average valley of 300 write request units per second and an average peak of 4300. Your pipeline's OCUs should be set with a minimum of 1 (300/1000 - 1, but at least 1) and a maximum of 5 (4300/1000 + 1).
- Follow best practices on scaling your destination OpenSearch Service indexes. If your indexes are under-scaled, it will slow down ingestion from DynamoDB, and might cause delays.

> **Note**
>
> GREATEST is a SQL function that, given a set of arguments, returns the argument with the greatest value.

Integrating DynamoDB with Amazon EventBridge

Amazon DynamoDB offers DynamoDB Streams for change data capture, enabling the capture of item-level changes in DynamoDB tables. DynamoDB Streams can invoke Lambda functions to process those changes, allowing event driven integration with other services and applications. DynamoDB Streams also supports filtering, which allows for efficient and targeted event processing.

DynamoDB Streams supports up to two simultaneous consumers per shard and supports filtering through Lambda event filtering so that only items which match specific criteria are processed. Some customers may have requirements to support more than two consumers. Others may need to enrich change events before they are processed, or use more advanced filtering and routing.

Integrating DynamoDB with EventBridge can support those requirements.

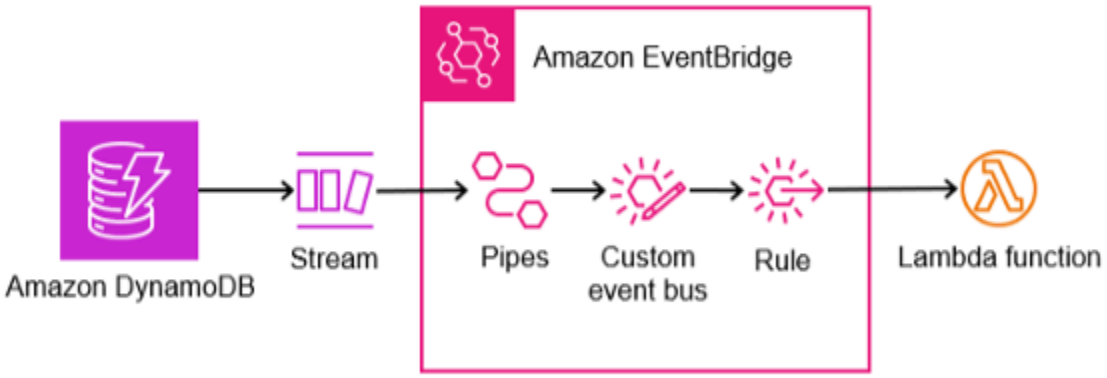
Amazon EventBridge is a serverless service that uses events to connect application components together, making it easier for you to build scalable event-driven applications. EventBridge offers native integration with Amazon DynamoDB through EventBridge Pipes, enabling seamless data flow from DynamoDB to an EventBridge bus. That bus can then fan-out to multiple applications and services through a set of rules and targets.

Topics

- How it works
- Creating an integration through the console
- Next steps

How it works

The integration between DynamoDB and EventBridge pipes uses DynamoDB Streams to capture a time-ordered sequence of item-level changes in a DynamoDB table. Each record captured this way contains the data modified in the table.



An EventBridge pipe consumes events from DynamoDB Streams and routes them to a target such as an EventBridge bus (an event bus is a router that receives events and delivers them to destinations, also called targets). Delivery is based on which rules match the contents of the event. Optionally, the pipe also includes the ability to filter for specific events and perform enrichments on the event data before sending it to the target.

While EventBridge supports multiple target types, a common choice when implementing a fan-out design is to use a Lambda function as the target. The following example demonstrates an integration with a Lambda function target.

Creating an integration through the console

Follow the steps below to create an integration through the AWS Management Console.

1. Enable DynamoDB Streams on the source table by following the steps in the Enabling a stream section of the DynamoDB developer guide. If DynamoDB Streams is already enabled on the source table, verify that there are currently fewer than two consumers. Consumers could be Lambda functions, DynamoDB Global Tables, Amazon DynamoDB zero-ETL integrations with Amazon OpenSearch Service, or applications that read directly from streams such as through the DynamoDB Streams Kinesis adapter.
2. Create an EventBridge event bus by following the steps in the Creating an Amazon EventBridge event bus section of the EventBridge user guide.
   a. When creating the event bus, enable **Schema discovery**.
3. Create an EventBridge pipe by following the steps in the Creating an Amazon EventBridge pipe section of the EventBridge user guide.

a. When configuring the source, in the **Source** field select *DynamoDB* and in the **DynamoDB Streams** field select the name of the source table stream.

b. When configuring the target, in the **Target service** field select *EventBridge event bus* and in the **Event bus as target** field select the event bus created in step 2.

4. Write an example item to the source DynamoDB table to trigger an event. This will allow EventBridge to infer schema from the example item. This schema can be used to create rules for routing events. For example, if you are implementing a design pattern that involves overloading attributes, you may want to trigger different rules depending on the value of your sort key. Details on how to write an item to DynamoDB can be found in the Working with items and attributes section of the DynamoDB developer guide.

5. Create an example Python Lambda function to be used as a target by following the steps in the Building Lambda functions with Python section of the Lambda developer guide. When creating your function, you can use the below example code to demonstrate the integration. When invoked, it will print the `NewImage` and `OldImage` received with the event which can be viewed in CloudWatch Logs.

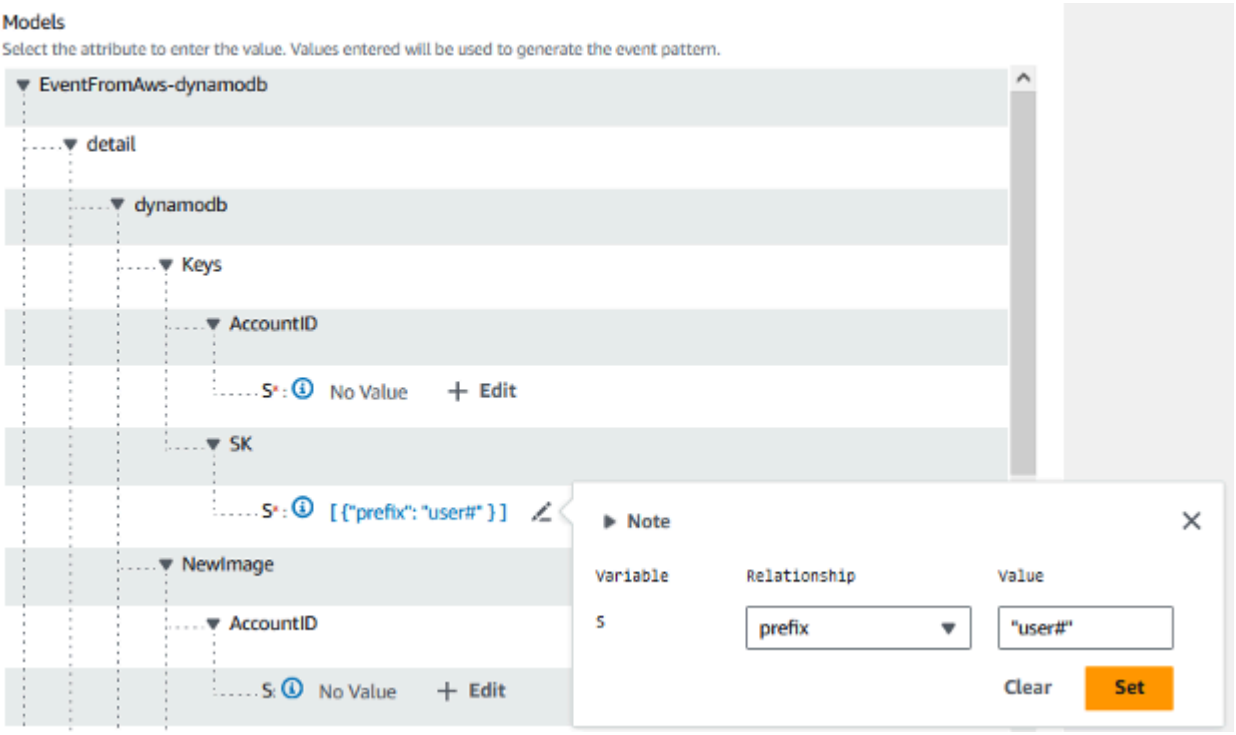
```
import json

def lambda_handler(event, context):
    dynamodb = event.get('detail', {}).get('dynamodb', {})
    new_image = dynamodb.get('NewImage')
    old_image = dynamodb.get('OldImage')

    if new_image:
        print("NewImage:", json.dumps(new_image, indent=2))
    if old_image:
        print("OldImage:", json.dumps(old_image, indent=2))

    return {'statusCode': 200, 'body': json.dumps(event)}
```

6. Create an EventBridge rule that will route events to your new Lambda function by following the steps in the Create a rule section that reacts to events EventBridge user guide.

a. When defining the rule detail, select the name of the event bus you created in step 2 as the **Event bus**.

b. When building the event pattern, follow the guide for **Existing schema**. Here, you can select the *discovered-schemas* registry and the discovered schema for your event. This allows you to configure an event pattern specific to your use case that only routes messages that match

specific attributes. For example, if you wanted to match only on DynamoDB items where the SK begins with "`user#`", you'd use a configuration like this.



c. Click **Generate event pattern in JSON** after you've finished designing a pattern against your schema. If you instead want to match all events that appear on DynamoDB Streams, use the following JSON for the event pattern.

```
{
  "source": ["aws.dynamodb"]
}
```

d. When selecting targets, follow the guide for AWS service. In the Select a target field, choose "Lambda function". In the Function field, select the Lambda function you created in step 5.

7. You can now stop schema discovery on your event bus by following the steps in the Starting or stopping schema discovery on event buses section of the EventBridge user guide.
8. Write a second example item to the source DynamoDB table to trigger an event. Validate that the event was successfully processed at each step.

   a. View the CloudWatch metric **PutEventsApproximateSuccessCount** for your event bus by following the Monitoring Amazon EventBridge section of the EventBridge user guide.

   b. View function logs for your Lambda function by following the Monitoring and troubleshooting Lambda functions section of the Lambda developer guide. If your Lambda

function uses the example code provided, you should see the `NewImage` and `OldImage` from DynamoDB Streams printed in the CloudWatch Logs log group.

c. View the **Error count and success rate (%)** metric for your Lambda function by following the Monitoring and troubleshooting Lambda functions section of the Lambda developer guide.

Next steps

This example provides a basic integration with a single Lambda function as a target. For a better understanding of more complex configurations, such as creating multiple rules, creating multiple targets, integrating with other services, and enriching events see the complete EventBridge user guide: Getting started with EventBridge.

> **Note**
>
> Be aware of any EventBridge quotas that might be relevant to your application. While DynamoDB Streams capacity scales with your table, EventBridge quotas are separate. Common quotas to be aware of in a large application would be **Invocations throttle limit in transactions per second** and **PutEvents throttle limit in transactions per second**. These quotas specify the number of invocations that can be sent to targets and the number of events that can be put into the bus per second.

Integrating DynamoDB with Amazon Managed Streaming for Apache Kafka

Amazon Managed Streaming for Apache Kafka (Amazon MSK) makes it easy to ingest and process streaming data in real time with a fully managed, highly available Apache Kafka service.

Apache Kafka is a distributed data store optimized for ingesting and processing streaming data in real-time. Kafka can process streams of records, effectively store streams of records in the order in which records were generated, and publish and subscribe to streams of records.

Because of these features, Apache Kafka is often used to build real-time streaming data pipelines. A *data pipeline* reliably processes and moves data from one system to another and can be an important part of adopting a purpose-built database strategy by facilitating the use of multiple databases which each support different use cases.

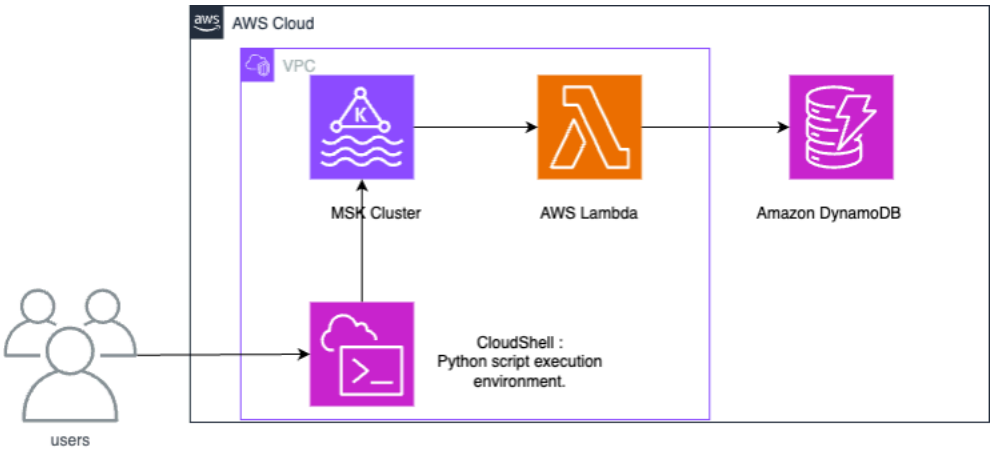
Amazon DynamoDB is common target in these data pipelines to support applications that use key-value or document data models and desire limitless scalability with consistent single-digit millisecond performance.

Topics

- How it works
- Set up an integration between Amazon MSK and DynamoDB
- Next steps

How it works

An Integration between Amazon MSK and DynamoDB uses a Lambda function to consume records from Amazon MSK and write them to DynamoDB.



Lambda internally polls for new messages from Amazon MSK and then synchronously invokes the target Lambda function. The Lambda function's event payload contains batches of messages from Amazon MSK. For the integration between Amazon MSK and DynamoDB, the Lambda function writes these messages to DynamoDB.

Set up an integration between Amazon MSK and DynamoDB

> **Note**
>
> You can download the resources used in this example at the following GitHub repository.

The steps below show how to set up a sample integration between Amazon MSK and Amazon DynamoDB. The example represents data generated by Internet of Things (IoT) devices and ingested into Amazon MSK. As data is ingested into Amazon MSK, it can be integrated with analytics services or third-party tools compatible with Apache Kafka, enabling various analytics use cases. Integrating DynamoDB as well provides key value lookup of individual device records.

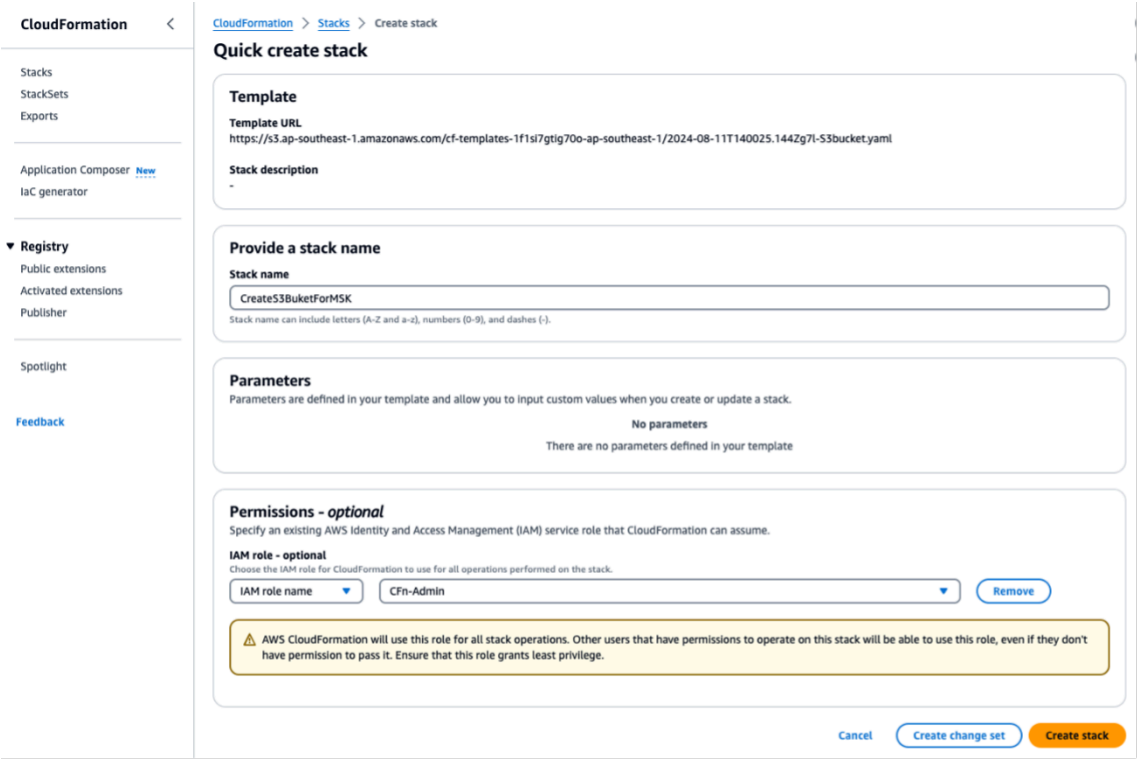
This example will demonstrate how a Python script writes IoT sensor data to Amazon MSK. Then, a Lambda function writes items with the partition key "`deviceid`" to DynamoDB.

The provided CloudFormation template will create the following resources: An Amazon S3 bucket, an Amazon VPC, a Amazon MSK cluster, and an AWS CloudShell for testing data operations.

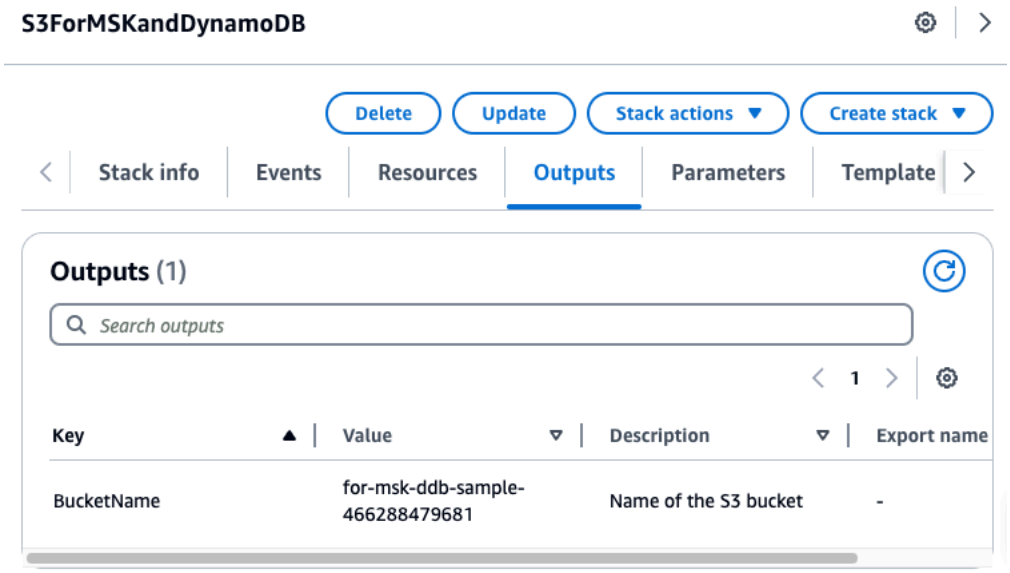
To generate test data, create an Amazon MSK topic and then create a DynamoDB table. You can use Session Manager from the management console to log into the CloudShell's operating system and run Python scripts.

After running the CloudFormation template, you can finish building this architecture by performing the following operations.

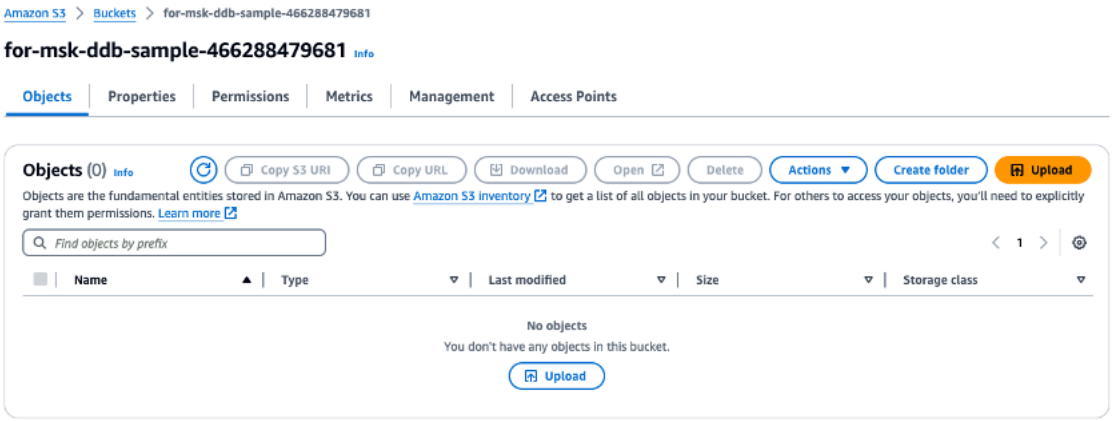
1. Run the CloudFormation template `S3bucket.yaml` to create an S3 bucket. For any subsequent scripts or operations, please run them in the same Region. Enter `ForMSKTestS3` as the CloudFormation stack name.



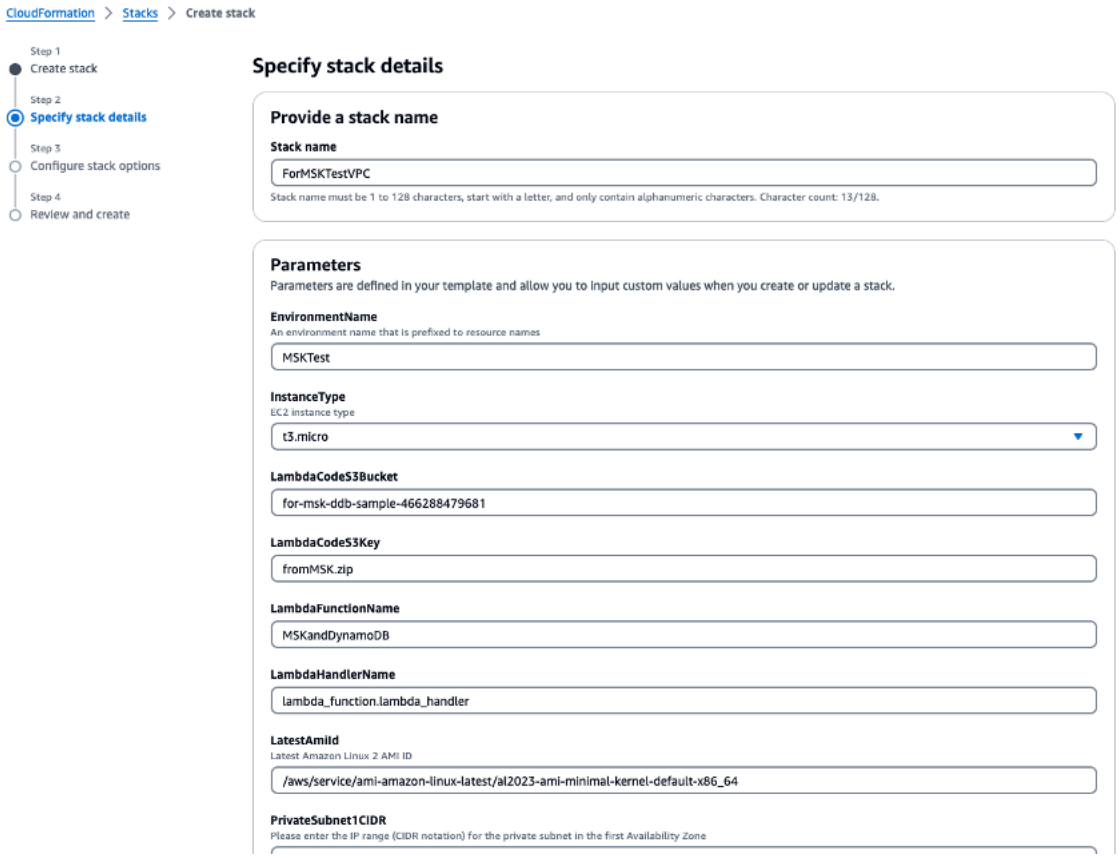
After this is completed, note down the S3 bucket name output under *Outputs*. You will need the name in Step 3.



2. Upload the downloaded ZIP file `fromMSK.zip` to the S3 bucket you just created.

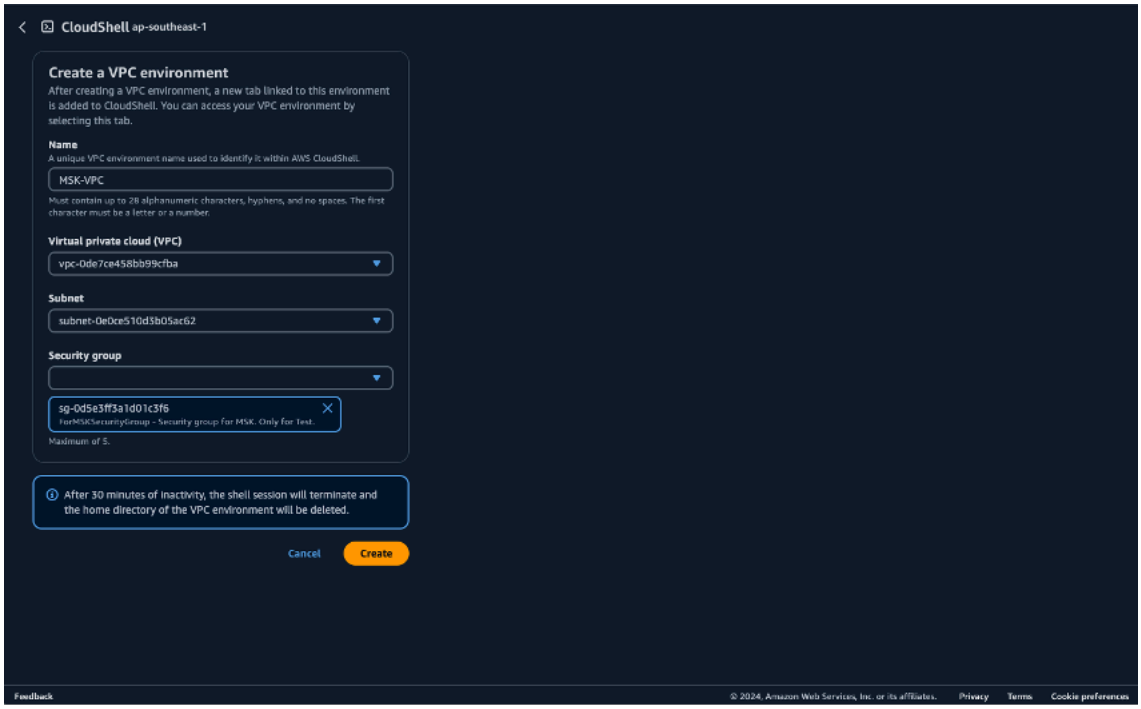


3. Run the CloudFormation template `VPC.yaml` to create a VPC, Amazon MSK cluster, and Lambda function. On the parameter input screen, enter the S3 bucket name you created in Step 1 where it asks for the S3 bucket. Set the CloudFormation stack name to `ForMSKTestVPC`.



4. Prepare the environment for running Python scripts in CloudShell. You can use CloudShell on the AWS Management Console. For more information on using CloudShell, see Getting started with AWS CloudShell. After starting CloudShell, create a CloudShell that belongs to the VPC you have just created in order to connect to the Amazon MSK Cluster. Create the CloudShell in a private subnet. Fill in the following fields:

    1. **Name** - can be set to any name. An example is **MSK-VPC**
    2. **VPC** - select **MSKTest**
    3. **Subnet** - select **MSKTest Private Subnet (AZ1)**
    4. **SecurityGroup** - select **ForMSKSecurityGroup**



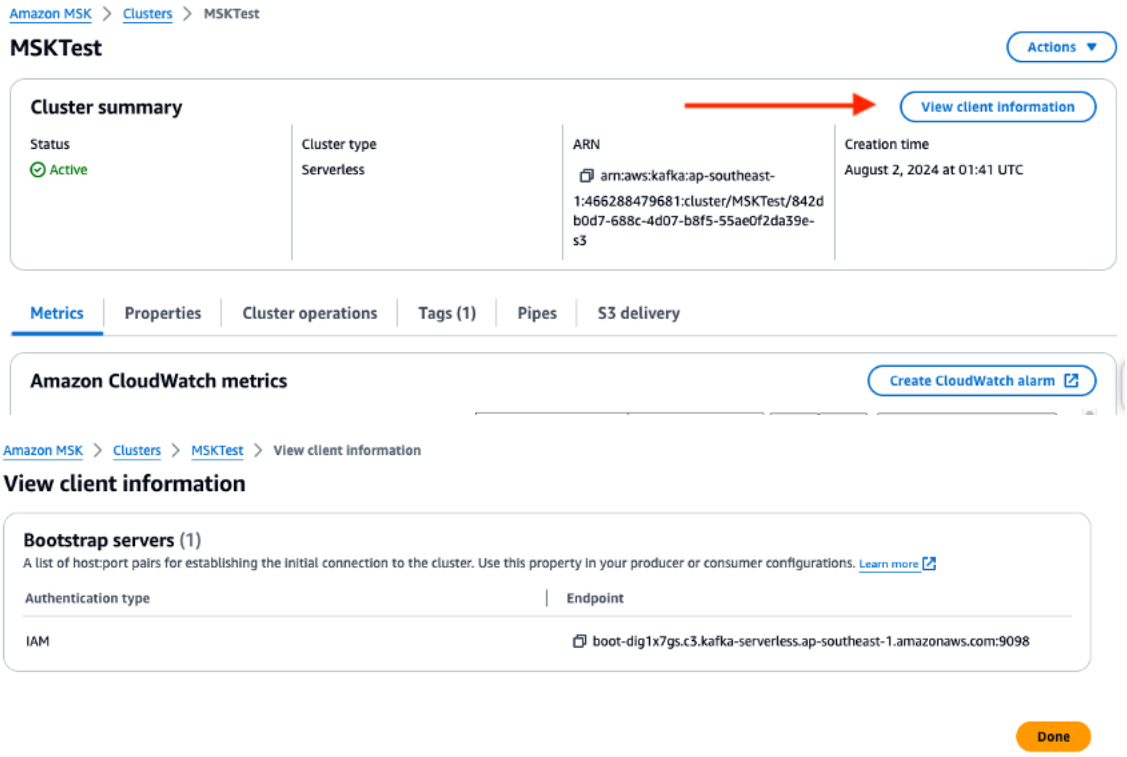
Once the CloudShell belonging to the Private Subnet has started, run the following command:

```
pip install boto3 kafka-python aws-msk-iam-sasl-signer-python
```

5. Download Python scripts from the S3 bucket.

```
aws s3 cp s3://[YOUR-BUCKET-NAME]/pythonScripts.zip ./
unzip pythonScripts.zip
```

6. Check the management console and set the environment variables for the broker URL and Region value in the Python scripts. Check the Amazon MSK cluster broker endpoint in the management console.



7. Set the environment variables on the CloudShell. If you are using the US West (Oregon):

```
export AWS_REGION="us-west-2"
export MSK_BROKER="boot-YOURMSKCLUSTER.c3.kafka-serverless.ap-
southeast-1.amazonaws.com:9098"
```

8. Run the following Python scripts.

   Create an Amazon MSK topic:

```
python ./createTopic.py
```

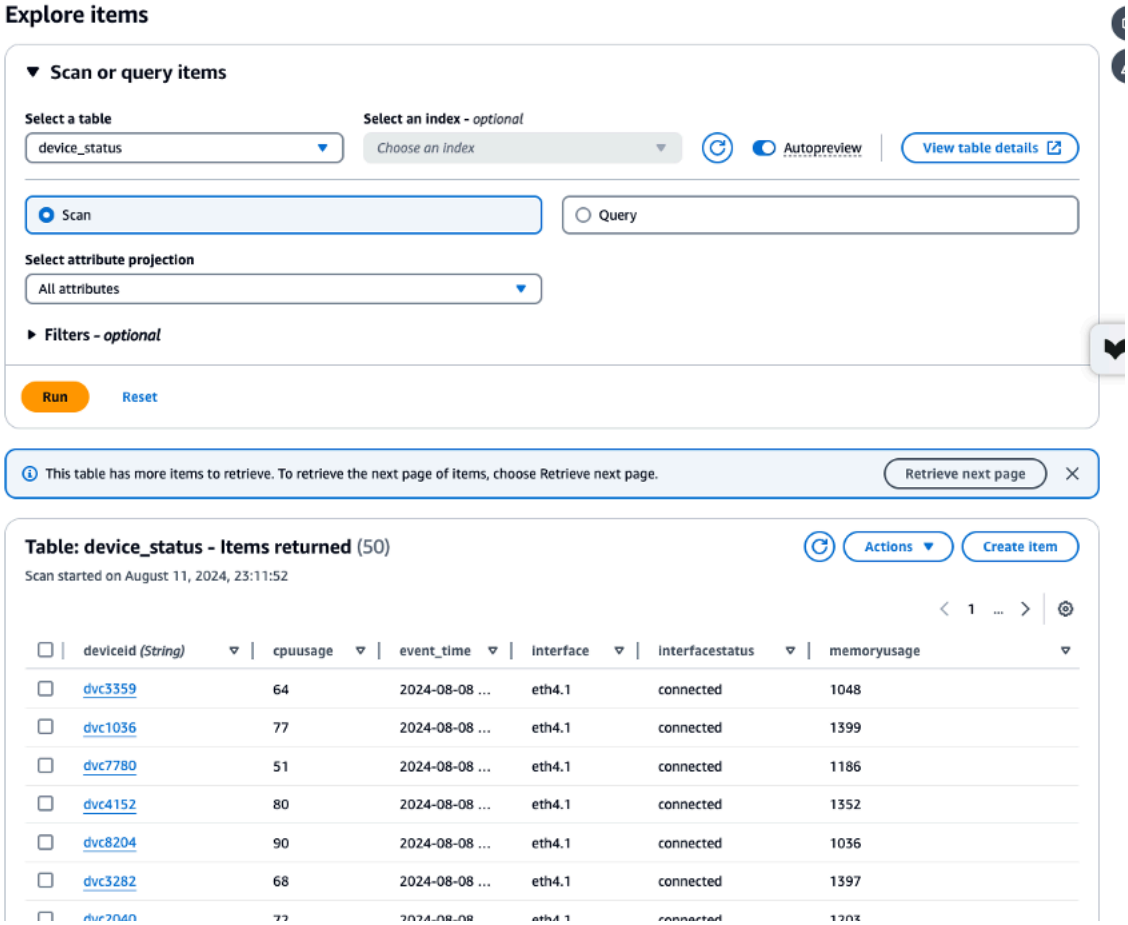
   Create a DynamoDB table:

```
python ./createTable.py
```

Write test data to the Amazon MSK topic:

```
python ./kafkaDataGen.py
```

9. Check the CloudWatch metrics for the created Amazon MSK, Lambda, and DynamoDB resources, and verify the data stored in the `device_status` table using the DynamoDB Data Explorer to ensure all processes ran correctly. If each process is run without error, you can check that the test data written from CloudShell to Amazon MSK is also written to DynamoDB.



10. When you're done with this example, delete the resources created in this tutorial. Delete the two CloudFormation stacks: `ForMSKTestS3` and `ForMSKTestVPC`. If the stack deletion completes successfully, all resources will be deleted.

Next steps

> **Note**
>
> If you created resources while following along with this example, please remember to delete them to avoid any unexpected charges.

The Integration identified an architecture that links Amazon MSK and DynamoDB to enable stream data to support OLTP workloads. From here, more complex searches can be realized by linking DynamoDB with OpenSearch Service. Consider integrating with EventBridge for more complex event-driven needs, and extensions such as Amazon Managed Service for Apache Flink for higher throughput and lower latency requirements.

Best practices for integrating with DynamoDB

When integrating DynamoDB with other services, you should always follow the best practices for using each individual service. However, there are some best practices specific to integration that you should consider.

Topics

- Creating a snapshot in DynamoDB
- Capturing data change in DynamoDB

Creating a snapshot in DynamoDB

- Generally, we recommend using export to Amazon S3 to create snapshots for initial replication. It is both cost effective, and won't compete with your application's traffic for throughput. You can also consider a backup and restore to a new table followed by a scan operation. This will avoid competing for throughput with your application, but will generally be substantially less cost effective than an export.
- Always set a `StartTime` when doing an export. This makes it easy to determine where you'll start your change data capture (CDC) from.
- When using export to S3, set a lifecycle action on the S3 bucket. Typically, an expiration action set at 7 days is safe, but you should follow any guidelines that your company might have. Even

if you explicitly delete your items after ingestion, this action can help catch issues, which helps reduce unnecessary costs and prevents policy violations.

Capturing data change in DynamoDB

- If you need near real-time CDC, use DynamoDB Streams or Amazon Kinesis Data Streams (KDS). When you're deciding which one to use, generally consider which is easiest to use with the downstream service. If you need to provide in-order event processing at a partition-key level, or if you have items that are exceptionally large, use DynamoDB Streams.
- If you don't need near real-time CDC, you can use export to Amazon S3 with incremental exports to export only the changes that have happened between two points in time.

  If you used export to S3 for generating a snapshot, this can be especially helpful because you can use similar code to process incremental exports. Typically, export to S3 is slightly cheaper than the previous streaming options, but cost is typically not the main factor for which option to use.
- You can generally only have two simultaneous consumers of a DynamoDB stream. Consider this when planning your integration strategy.
- Don't use scans to detect changes. This might work on a small scale, but becomes impractical fairly quickly.

Using agentic AI with DynamoDB

Amazon DynamoDB is a serverless, fully managed, distributed NoSQL database with single-digit millisecond performance at any scale. DynamoDB is optimized for high-throughput workloads and you can extend its capabilities by integrating with generative AI models. Using generative AI models, you can work with data stored in DynamoDB tables in real-time and build applications that are contextually aware and highly personalized. You can also enhance the end user experience by fully leveraging your business, user, and application data to customize your generative AI solutions.

For more information about gen AI and the solutions AWS provides to build gen AI applications, see Transform your business with generative AI.

Topics

- Generative AI use cases for DynamoDB
- Generative AI blogs for DynamoDB
- Leveraging DynamoDB Zero-ETL integration with OpenSearch Service
- Using DynamoDB as a checkpoint store for LangGraph agents

Generative AI use cases for DynamoDB

DynamoDB is widely used in AI powered conversational applications, such as chatbots and call centers built with a Foundation Model (FM). You can access FMs through Amazon Bedrock, Amazon SageMaker AI, or other model providers. Such applications commonly use DynamoDB to improve personalization and enhance the user experience across three data patterns: application data, business data, and user data. Some examples of these data patterns are as follows:

- Storage of application data, such as chat message history, through integrations with LangChain, LlamaIndex, or a custom code. This context enhances the user experience by allowing the model to *converse* back and forth with the user.
- Creation of a customized user experience by leveraging business data, such as inventory, pricing, and documentation.
- Application of user data, such as web history, past orders, and user preferences, to provide personalized answers.

For instance, an insurance company can build a chatbot using DynamoDB to provide their Retrieval-Augmented Generation (RAG) based gen AI model access to near real-time data. Examples of such data are real-time mortgage rates, product pricing, compliant/standard contract copy, user web history, and user preferences. Combining DynamoDB with RAG adds in-depth and updated information about insurance products and the user data. This enriches the prompts and answers to provide end users with an accurate, personalized, and near real-time experience.

Similarly, financial services industry customers use DynamoDB, Amazon Bedrock knowledge bases, and Amazon Bedrock agents to build RAG-based gen AI applications. These applications can use open-source earnings reports and call transcripts. They can also use user-specific portfolio and transaction history to generate an on-demand summary of portfolio including an outlook for the future.

Generative AI blogs for DynamoDB

The following articles offer detailed use cases, best practices, and step-by-step guides to help you leverage DynamoDB's capabilities in building advanced AI-powered applications.

- Amazon DynamoDB data models for generative AI chatbots
- Build a scalable, context-aware chatbot with Amazon DynamoDB, Amazon Bedrock, and LangChain
- Build durable AI agents with LangGraph and Amazon DynamoDB

Leveraging DynamoDB Zero-ETL integration with OpenSearch Service

You can use Amazon Bedrock with DynamoDB to provide serverless access to foundational models (FMs), such as Amazon Titan and other third-party models. You can leverage the Zero-ETL integration with Amazon OpenSearch Service to enable vector search capabilities when building generative AI applications. The Generative AI with DynamoDB zero-ETL to OpenSearch integration and Amazon Bedrock workshop provides you hands-on experience in setting up DynamoDB Zero-ETL integration with OpenSearch. This workshop does the following tasks:

- Creates a pipeline from your DynamoDB table to OpenSearch.
- Creates an Amazon Bedrock Connector in OpenSearch.
- Queries Amazon Bedrock leveraging OpenSearch as a vector store.

- Uses the Claude FM in Amazon Bedrock to create a written response in plain English explaining the search results returned by OpenSearch.

This workshop enables you to integrate DynamoDB with OpenSearch to build generative AI applications. It also demonstrates the flexible querying capability across database engines to help you integrate DynamoDB and OpenSearch for traditional use cases. This workshop is one of the seven modules in the Amazon DynamoDB Immersion Day. You can run this workshop in any AWS account.

You can also refer to the following blog post about how to set up a Zero-ETL integration between DynamoDB and OpenSearch Service. This blog post also describes how to set up model connectors in OpenSearch Service to automatically generate embeddings using Amazon Bedrock for incoming data. Vector search for Amazon DynamoDB with zero ETL for Amazon OpenSearch Service.

Using DynamoDB as a checkpoint store for LangGraph agents

LangGraph is a framework for building stateful, multi-actor AI applications with Large Language Models (LLMs). LangGraph agents require persistent storage to maintain conversation state, enable human-in-the-loop workflows, support fault tolerance, and provide time-travel debugging capabilities. DynamoDB's serverless architecture, single-digit millisecond latency, and automatic scaling make it an ideal checkpoint store for production LangGraph deployments on AWS.

The `langgraph-checkpoint-aws` package provides a `DynamoDBSaver` class that implements the LangGraph checkpoint interface, enabling you to persist agent state in DynamoDB with optional Amazon Simple Storage Service offloading for large checkpoints.

Key features

State persistence

: Automatically saves agent state after each step, enabling agents to resume from interruptions and recover from failures.

Time to Live-based cleanup

: Automatically expire old checkpoints using DynamoDB Time to Live to manage storage costs.

Compression

: Optionally compress checkpoint data with gzip to reduce storage costs and improve throughput.

Amazon S3 offloading

Automatically offload large checkpoints (greater than 350 KB) to Amazon Simple Storage Service to work within DynamoDB item size limits.

Sync and async support

Both synchronous and asynchronous APIs for flexibility in different application architectures.

Prerequisites

- Python 3.10 or later
- An AWS account with permissions to create DynamoDB tables (and optionally Amazon S3 buckets)
- AWS credentials configured (see the AWS documentation for credential setup options)

> **Important**
>
> This guide creates AWS resources that may incur charges. DynamoDB uses pay-per-request billing by default, and Amazon S3 charges apply if you enable large checkpoint offloading. Follow the Clean up section to delete resources when you are done.

Installation

Install the checkpoint package from PyPI:

```
pip install langgraph-checkpoint-aws
```

Basic usage

The following example demonstrates how to configure DynamoDB as a checkpoint store for a LangGraph agent:

```
from langgraph.graph import StateGraph
from langgraph_checkpoint_aws import DynamoDBSaver
from typing import TypedDict

# Define your state schema
```

```
class State(TypedDict):
    input: str
    result: str

# Initialize the DynamoDB checkpoint saver
checkpointer = DynamoDBSaver(
    table_name="langgraph-checkpoints",
    region_name="us-east-1"
)

# Build your LangGraph workflow
builder = StateGraph(State)
builder.add_node("process", lambda state: {"result": "processed"})
builder.set_entry_point("process")
builder.set_finish_point("process")

# Compile the graph with the DynamoDB checkpointer
graph = builder.compile(checkpointer=checkpointer)

# Invoke the graph with a thread ID to enable state persistence
config = {"configurable": {"thread_id": "session-123"}}
result = graph.invoke({"input": "data"}, config)
```

The `thread_id` in the configuration acts as the partition key in DynamoDB, allowing you to maintain separate conversation threads and retrieve historical states for any thread.

Production configuration

For production deployments, you can enable Time to Live, compression, and Amazon S3 offloading. You can also use the `endpoint_url` parameter to point to a local DynamoDB instance for testing:

```
import boto3
from botocore.config import Config
from langgraph_checkpoint_aws import DynamoDBSaver

# Production configuration
session = boto3.Session(
    profile_name="production",
    region_name="us-east-1"
)

checkpointer = DynamoDBSaver(
    table_name="langgraph-checkpoints",
```

```
    session=session,
    ttl_seconds=86400 * 7,           # Expire checkpoints after 7 days
    enable_checkpoint_compression=True,  # Enable gzip compression
    boto_config=Config(
        retries={"mode": "adaptive", "max_attempts": 6},
        max_pool_connections=50
    ),
    s3_offload_config={
        "bucket_name": "my-checkpoint-bucket"
    }
)

# Local testing with DynamoDB Local
local_checkpointer = DynamoDBSaver(
    table_name="langgraph-checkpoints",
    region_name="us-east-1",
    endpoint_url="http://localhost:8000"
)
```

DynamoDB table configuration

The checkpoint saver requires a DynamoDB table with a composite primary key. You can create the table using the following AWS CloudFormation template:

```
AWSTemplateFormatVersion: '2010-09-09'
Description: 'DynamoDB table for LangGraph checkpoint storage'

Parameters:
  TableName:
    Type: String
    Default: langgraph-checkpoints

Resources:
  CheckpointTable:
    Type: AWS::DynamoDB::Table
    DeletionPolicy: Retain
    UpdateReplacePolicy: Retain
    Properties:
      TableName: !Ref TableName
      BillingMode: PAY_PER_REQUEST
      AttributeDefinitions:
        - AttributeName: PK
          AttributeType: S
```

```
        - AttributeName: SK
          AttributeType: S
      KeySchema:
        - AttributeName: PK
          KeyType: HASH
        - AttributeName: SK
          KeyType: RANGE
      TimeToLiveSpecification:
        AttributeName: ttl
        Enabled: true
      PointInTimeRecoverySpecification:
        PointInTimeRecoveryEnabled: true
      SSESpecification:
        SSEEnabled: true
```

Deploy the template with the AWS CLI:

```
aws cloudformation deploy \
  --template-file template.yaml \
  --stack-name langgraph-checkpoint \
  --parameter-overrides TableName=langgraph-checkpoints
```

Required IAM permissions

The following IAM policy provides the minimum permissions required for the DynamoDB checkpoint saver. Replace *111122223333* with your AWS account ID and update the Region to match your environment.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "dynamodb:GetItem",
        "dynamodb:PutItem",
        "dynamodb:Query",
        "dynamodb:BatchGetItem",
        "dynamodb:BatchWriteItem"
      ],
      "Resource": "arn:aws:dynamodb:us-east-1:111122223333:table/langgraph-checkpoints"
    }
```

```
  ]
}
```

If you enable Amazon S3 offloading, add the following statement to the policy:

```
{
  "Effect": "Allow",
  "Action": [
    "s3:PutObject",
    "s3:GetObject",
    "s3:DeleteObject",
    "s3:PutObjectTagging"
  ],
  "Resource": "arn:aws:s3:::my-checkpoint-bucket/*"
},
{
  "Effect": "Allow",
  "Action": [
    "s3:GetBucketLifecycleConfiguration",
    "s3:PutBucketLifecycleConfiguration"
  ],
  "Resource": "arn:aws:s3:::my-checkpoint-bucket"
}
```

Asynchronous usage

For asynchronous applications, use the async methods provided by the checkpoint saver:

```
import asyncio
from langgraph.graph import StateGraph
from langgraph_checkpoint_aws import DynamoDBSaver
from typing import TypedDict

class State(TypedDict):
    input: str
    result: str

async def main():
    checkpointer = DynamoDBSaver(
        table_name="langgraph-checkpoints",
        region_name="us-east-1"
    )
    builder = StateGraph(State)
```

```
    builder.add_node("process", lambda state: {"result": "processed"})
    builder.set_entry_point("process")
    builder.set_finish_point("process")
    graph = builder.compile(checkpointer=checkpointer)

    config = {"configurable": {"thread_id": "async-session-123"}}
    result = await graph.ainvoke({"input": "data"}, config)
    return result

asyncio.run(main())
```

Clean up

To avoid ongoing charges, delete the resources you created:

```
# Delete the DynamoDB table
aws dynamodb delete-table --table-name langgraph-checkpoints

# Delete the CloudFormation stack (if you used the template above)
aws cloudformation delete-stack --stack-name langgraph-checkpoint

# If you created an S3 bucket for large checkpoint offloading, empty and delete it
aws s3 rm s3://my-checkpoint-bucket --recursive
aws s3 rb s3://my-checkpoint-bucket
```

Error handling

Common error scenarios:

- **Table not found**: Verify the `table_name` and `region_name` match your DynamoDB table.
- **Throttling**: If you see `ProvisionedThroughputExceededException`, consider switching to on-demand billing mode or increasing provisioned capacity.
- **Item size exceeded**: If checkpoints exceed 350 KB, enable Amazon S3 offloading (see Production configuration).
- **Credential errors**: Verify your AWS credentials are valid and have the required permissions.

Additional resources

- langgraph-checkpoint-aws on PyPI

- langgraph-checkpoint-aws on GitHub
- LangGraph documentation
- DynamoDB best practices
- Build durable AI agents with LangGraph and Amazon DynamoDB

Quotas and constraints for Amazon DynamoDB

This topic describes current quotas, formerly referred to as limits, within Amazon DynamoDB. This topic also describes how you can perform the quota management tasks, for example, viewing your current quotas and requesting a quota increase.

Topics

- Performing quota management tasks in DynamoDB
- Requesting a quota increase in DynamoDB
- Quotas in Amazon DynamoDB
- Constraints in Amazon DynamoDB

Performing quota management tasks in DynamoDB

Amazon DynamoDB has several service components, such as tables, streams, indexes, and more. When you create your AWS account, there are default *quotas* (formerly referred to as limits) set on these components. Unless otherwise noted, each quota is Region-specific. You can request increases for some of the quotas. After a quota for a resource has been reached, additional requests to create that resource fail with an exception.

Accessing DynamoDB quotas

You can work with DynamoDB Service Quotas in the following ways:

- AWS Management Console

  The Service Quotas console is a browser-based interface that you can use to view and manage your Service Quotas. You can access Service Quotas from any AWS Management Console page by choosing it on the top navigation bar, or by searching for Service Quotas in the AWS Management Console.
- AWS Command Line Interface tools

  When using AWS Command Line Interface tools, you can issue commands at your system's command line to perform Service Quotas tasks. The command line tools are useful if you want to build scripts that perform AWS tasks.
- AWS SDKs

You can use the AWS SDKs for various programming languages and platforms (for example, Java, Python, Ruby, .NET, iOS and Android, and others) to perform Service Quotas tasks.

If an adjustable quota isn't available in the Service Quotas console, use the AWS Support Center Console to create a Service Quotas increase case.

Viewing current quotas in the console

To view your current DynamoDB quotas using the Service Quotas console

1. Open the Service Quotas console at https://console.aws.amazon.com/servicequotas/home/services/dynamodb/quotas/
2. From the navigation bar, at the top of the screen, select a Region.
3. The console displays details about the DynamoDB **Quota name**, **Applied account-level quota value**, **AWS default quota value**, **Utilization**, and the **Adjustability** of the quota at the account-level or resource-level.

   If the applied quota value or utilization is not available, the console displays **Not available**. You can request your applied quota value through the Support Center Console.
4. Choose a specific **Quota name** to view the **Details** page, which displays that quota's **Description**, **Quota code**, **Quota ARN**, **Utilization**, **Applied account-level quota value**, **Adjustability**, and **AWS default quota value**.

   If applicable, the **Details** page also displays any **Monitoring** options, **Alarms**,**request history**, and any of the quota's **Tags**.

Viewing current quotas using the AWS CLI

To view the default values for DynamoDB quotas:

- Call the `ListDefaultServiceQuotas` operation with the DynamoDB service code (dynamodb) to retrieve default values for Amazon DynamoDB Service quotas.

```
$ aws service-quotas list-aws-default-service-quotas \
        --service-code dynamodb

 {
```

```
    "Quotas": [
        {
            "ServiceCode": "dynamodb",
            "ServiceName": "Amazon DynamoDB",
            "QuotaArn": "arn:aws:servicequotas:us-east-1::dynamodb/L-F7858A77",
            "QuotaCode": "L-F7858A77",
            "QuotaName": "Global Secondary Indexes per table",
            "Value": 20.0,
            "Unit": "None",
            "Adjustable": true,
            "GlobalQuota": false
        },
        {
            "ServiceCode": "dynamodb",
            "ServiceName": "Amazon DynamoDB",
            "QuotaArn": "arn:aws:servicequotas:us-east-1::dynamodb/L-AB614373",
            "QuotaCode": "L-AB614373",
            "QuotaName": "Table-level write throughput limit",
            "Value": 40000.0,
            "Unit": "None",
            "Adjustable": true,
            "GlobalQuota": false
        }......
    ]
}
```

To view the applied quota values:

- Call the `ListServiceQuotas` operation with the DynamoDB service code (dynamodb) to retrieve all applied quota values either at the account-level, resource-level, or all levels by passing `ACCOUNT`, `RESOURCE`, or `ALL` respectively as the value for the parameter `QuotaAppliedAtLevel`. The following CLI example retrieves quota values applied at the account-level.

```
$ aws service-quotas list-service-quotas \
        --service-code dynamodb \
        --quota-applied-at-level ACCOUNT


{
    "Quotas": [
```

```
        {
            "ServiceCode": "dynamodb",
            "ServiceName": "Amazon DynamoDB",
            "QuotaArn": "arn:aws:servicequotas:us-east-1:303935678045:dynamodb/L-
F7858A77",
            "QuotaCode": "L-F7858A77",
            "QuotaName": "Global Secondary Indexes per table",
            "Value": 20.0,
{
    "Quotas": [
        {
            "ServiceCode": "dynamodb",
            "ServiceName": "Amazon DynamoDB",
            "QuotaArn": "arn:aws:servicequotas:us-east-1:303935678045:dynamodb/L
-F7858A77",
            "QuotaCode": "L-F7858A77",
            "QuotaName": "Global Secondary Indexes per table",
            "Value": 20.0,
            "Unit": "None",
            "Adjustable": true,
            "GlobalQuota": false,
            "QuotaAppliedAtLevel": "ACCOUNT"
        }.....

        }
    ]
}
```

Requesting a quota increase in DynamoDB

You can request a quota increase for each Region using the Service Quotas console, AWS CLI or a support case. If an adjustable quota isn't available in the Service Quotas console, use the AWS Support Center Console to create a service quota increase case.

Support could approve, deny, or partially approve your quota increase requests. Increases aren't granted immediately, and can take a few days to take effect.

To request an increase using the Service Quotas console

1. Open the Service Quotas console at https://console.aws.amazon.com/servicequotas/home/services/dynamodb/quotas/

2. From the navigation bar, at the top of the screen, select a Region.
3. Filter the list by resource name. For example, enter **On-Demand** to locate the quotas for On-Demand Instances.
4. If the quota is adjustable, choose the quota and then choose **Request quota increase**.
5. For **Change quota value**, enter the new quota value.
6. Choose **Request**.
7. To view any pending or recently resolved requests in the console, choose **Dashboard** from the navigation pane. For pending requests, choose the status of the request to open the request receipt. The initial status of a request is **Pending**. After the status changes to **Quota requested**, you'll see the case number with Support. Choose the case number to open the ticket for your request.

For more information, including how to use the AWS CLI or SDKs to request a quota increase, see Requesting a quota increase in the *Service Quotas User Guide*.

Quotas in Amazon DynamoDB

This section describes current quotas, formerly referred to as limits, within Amazon DynamoDB. Each quota applies on a per-Region basis unless otherwise specified.

> **Note**
>
> All size measurements in DynamoDB use binary-based units. DynamoDB denotes 1 KB = 1024 bytes, 1 MB = 1024 KB, 1 GB = 1024 MB, 1 TB = 1024 GB.

Topics

- Read/write throughput
- Reserved Capacity
- Tables
- Global tables
- Secondary indexes
- Projected secondary index attributes
- DynamoDB Streams

- Import from Amazon S3
- Table export to Amazon S3
- Backup and restore
- Contributor Insights

Read/write throughput

Throughput default quotas

AWS places some default quotas on the throughput that your account can provision and consume within a Region.

The account-level read throughput and account-level write throughput quotas apply at the account level. These account-level quotas apply to the sum of the provisioned throughput capacity for all your account's tables and global secondary indexes in a given Region. All the account's available throughput can be provisioned for a single table or across multiple tables. These quotas only apply to tables using the provisioned capacity mode.

The table-level read throughput and table-level write throughput quotas apply differently to tables that use the provisioned capacity mode, and tables that use the on-demand capacity mode.

For provisioned capacity mode tables and GSIs, the quota is the maximum amount of read and write capacity units that can be provisioned for any table or any of its GSIs in the Region. The total of any individual table and all its GSIs must also remain below the account-level read and write throughput quota. This is in addition to the requirement that the total of all provisioned tables and their GSIs must remain below the account-level read and write throughput quota.

For on-demand capacity mode tables and GSIs, the table-level quota is the maximum read and write capacity units that are available for any table, or any individual GSI within that table. No account-level read and write throughput quotas are applied to tables in on-demand mode.

Following are the throughput quotas that apply on your account, by default.

> **Note**
>
> All capacity unit and request unit quotas are measured per second. For example, a quota of 40,000 read capacity units means 40,000 reads per second.

> **Note**
>
> You can request any number of read capacity units (RCU) or write capacity units (WCU) for your DynamoDB tables through a service quota increase. The values listed in the following table represent the initial default quotas. These are not maximum limits for your tables.

Throughput quota name	On-Demand	Provisioned	Adjustable
Per table	40,000 read request units and 40,000 write request units	40,000 read capacity units and 40,000 write capacity units	Yes
Per account	Not applicable	80,000 read capacity units and 80,000 write capacity units	Yes
Minimum throughput for any table or global secondary index	Not applicable	1 read capacity unit and 1 write capacity unit	Yes

Increasing or decreasing throughput (for provisioned tables)

Increasing provisioned throughput

You can increase `ReadCapacityUnits` or `WriteCapacityUnits` as often as necessary, using the AWS Management Console or the `UpdateTable` operation. In a single call, you can increase the provisioned throughput for a table, for any global secondary indexes on that table, or for any combination of these. The new settings do not take effect until the `UpdateTable` operation is complete.

You can't exceed your per-account quotas when you add provisioned capacity, and DynamoDB doesn't allow you to increase provisioned capacity very rapidly. Aside from these restrictions, you can increase the provisioned capacity for your tables as high as you need. For more information about per-account quotas, see the preceding section, Throughput default quotas.

Decreasing provisioned throughput

For every table and global secondary index in an `UpdateTable` operation, you can decrease `ReadCapacityUnits` or `WriteCapacityUnits` (or both). The new settings don't take effect until the `UpdateTable` operation is complete.

There is a default quota on the number of provisioned capacity decreases you can perform on your DynamoDB table per day. A day is defined according to Universal Time Coordinated (UTC). You start each day with 4 available decreases. Each hour, 1 additional decrease becomes available, up to the maximum of 4 available at any time. Over a full 24-hour day, this allows you to decrease up to 27 times (4 in the first hour, plus 1 for each of the remaining 23 hours).

> **Important**
>
> Table and global secondary index decrease limits are decoupled, so any global secondary indexes for a particular table have their own decrease limits. However, if a single request decreases the throughput for a table and a global secondary index, it is rejected if either exceeds the current limits. Requests are not partially processed.

Example

In the first 4 hours of a day, a table with a global secondary index can be modified as follows:

- Decrease the table's `WriteCapacityUnits` or `ReadCapacityUnits` (or both) four times.
- Decrease the `WriteCapacityUnits` or `ReadCapacityUnits` (or both) of the global secondary index four times.

At the end of that same day, the table and the global secondary index throughput can potentially be decreased a total of 27 times each.

Reserved Capacity

AWS places a default quota on the amount of active reserved capacity that your account can purchase. The quota limit is a combination of reserved capacity for write capacity units (WCUs) and read capacity units (RCUs).

Reserved capacity quota	Active reserved capacity	Adjustable
Per account	1,000,000 provisioned capacity units (WCUs _ RCUs)	Yes

If you attempt to purchase more than 1,000,000 provisioned capacity units in a single purchase, you will receive an error for this service quota limit. If you have active reserved capacity and attempt to purchase additional reserved capacity that would result in more than 1,000,000 active provisioned capacity units, you will receive an error for this service quota limit.

Tables

Table size

There is no practical limit on a table's size. Tables are unconstrained in terms of the number of items or the number of bytes.

Maximum number of tables per account per region

For any AWS account, there is an initial quota of 2,500 tables per AWS Region.

If you need more than 2,500 tables for a single account, please reach out to your AWS account team to explore an increase up to a maximum of 10,000 tables. For more than 10,000, the recommended best practice is to setup multiple accounts, each of which can serve up to 10,000 tables.

Global tables

The following default quotas apply when using global tables.

Default global table quotas	On-Demand	Provisioned
Number of MRSC global tables (See the section called "Consistency modes")	400 total MRSC global tables in any capacity mode	400 total MRSC global tables in any capacity mode

Default global table quotas	On-Demand	Provisioned
Throughput per table configured for multi-Region eventual consistency (MREC)	40,000 read request units and 40,000 write request units	40,000 read capacity units and 40,000 write capacity units
Throughput per table configured for multi-Region strong consistency (MRSC)	40,000 read request units and 40,000 write request units	40,000 read capacity units and 40,000 write capacity units
Backfilled data for new replicas per account, per Region, per day	10 TB	10 TB

Note

There may be instances where you will need to request a quota limit increase through AWS Support. If any of the following apply to you, please see https://aws.amazon.com/support:

- Global table throughput quotas must be equal to or greater than per table throughput quotas for replica creation to succeed. There are separate global table throughput quotas for MREC and MRSC global tables.
- If you are adding a replica or replicas to one destination Region within a 24-hour period with a combined total greater than 10TB, you must request a service quota increase for your add replica data backfill quota.
- If you encounter an error similar to the following:
  - Cannot create a replica of table 'example_table' in region 'example_region_A' because its exceeds your current account limit in region 'example_region_B'.

Secondary indexes

You can define up to 5 local secondary indexes per table.

There is a default quota of 20 global secondary indexes per table.

Projected secondary index attributes

You can project up to 100 attributes combined for all of a table's local and global secondary indexes. This quota only applies to user-specified projected attributes.

For the `CreateTable` operation, if you specify a `ProjectionType` of `INCLUDE`, the total count of attributes specified in `NonKeyAttributes` summed across all secondary indexes must not exceed 100. Projecting the same attribute name into two different indexes counts as two distinct attributes toward the quota.

This quota doesn't apply to secondary indexes with a `ProjectionType` of `KEYS_ONLY` or `ALL`.

DynamoDB Streams

Simultaneous readers of a shard in DynamoDB Streams

For single-Region tables that are not global tables, you can design for up to two simultaneous processes to read from the same DynamoDB Streams shard at the same time. Exceeding this limit can result in request throttling. For global tables, we recommend you limit the number of simultaneous readers to one to avoid request throttling.

Maximum write capacity for a table with DynamoDB Streams enabled

AWS places some default quotas on the write capacity for DynamoDB tables with DynamoDB Streams enabled. These default quotas are applicable only for tables in provisioned read/write capacity mode.

- Per table – 40,000 write capacity units

Import from Amazon S3

DynamoDB Import from Amazon S3 can support up to 50 concurrent import jobs with a total import source object size of 15TB at a time in us-east-1, us-west-2, and eu-west-1 regions. In all other regions, up to 50 concurrent import tasks with a total size of 1TB is supported. Each import job can take up to 50,000 Amazon S3 objects in all regions. For more information on import and validation, see import format quotas and validation.

Table export to Amazon S3

Full export: up to 300 concurrent export tasks, or up to a total of 100TB from all in-flight table exports, can be exported. Both of these limits are checked before an export is queued.

Incremental export: DynamoDB Incremental Export to Amazon S3 can support up to 300 concurrent export jobs or up to a total of 100TB from all in-flight table exports. The export period window limits are 15 minutes minimum and 24 hours maximum.

Backup and restore

DynamoDB supports up to 50 concurrent restores totaling 50 TB via DynamoDB on-demand or continuous backups. AWS Backup supports up to 50 concurrent restores totaling 25 TB.

Contributor Insights

When you enable Customer Insights on your DynamoDB table, you're still subject to Contributor Insights rules limits. For more information, see CloudWatch service quotas.

Constraints in Amazon DynamoDB

This section describes current constraints, formerly referred to as limits, within Amazon DynamoDB.

> **Note**
>
> All size measurements in DynamoDB use binary-based units. DynamoDB denotes 1 KB = 1024 bytes, 1 MB = 1024 KB, 1 GB = 1024 MB, 1 TB = 1024 GB.

Topics

- Read/write capacity mode
- Secondary indexes
- Partition keys and sort keys
- Naming rules
- Data types

- Items
- Attributes
- Expression parameters
- DynamoDB transactions
- DynamoDB Streams
- DynamoDB Accelerator (DAX)
- API-specific constraints
- DynamoDB encryption at rest

Read/write capacity mode

You can switch tables from provisioned capacity mode to on-demand mode up to four times in a 24-hour rolling window. You can switch tables from on-demand mode to provisioned capacity mode at any time.

For more information about switching between read and write capacity modes, see Considerations when switching capacity modes in DynamoDB.

Capacity unit sizes (for provisioned tables)

One read capacity unit = one strongly consistent read per second, or two eventually consistent reads per second, for items up to 4 KB in size.

One write capacity unit = one write per second, for items up to 1 KB in size.

Transactional read requests require two read capacity units to perform one read per second for items up to 4 KB.

Transactional write requests require two write capacity units to perform one write per second for items up to 1 KB.

Request unit sizes (for on-demand tables)

One read request unit = one strongly consistent read per second, or two eventually consistent reads per second, for items up to 4 KB in size.

One write request unit = one write per second, for items up to 1 KB in size.

Transactional read requests require two read request units to perform one read per second for items up to 4 KB.

Transactional write requests require two write request units to perform one write per second for items up to 1 KB.

Secondary indexes

Projected Secondary Index attributes per table

You can project a total of up to 100 attributes into all of a table's local and global secondary indexes. This only applies to user-specified projected attributes.

In a `CreateTable` operation, if you specify a `ProjectionType` of `INCLUDE`, the total count of attributes specified in `NonKeyAttributes`, summed across all of the secondary indexes, must not exceed 100. If you project the same attribute name into two different indexes, this counts as two distinct attributes when determining the total.

This limit does not apply for secondary indexes with a `ProjectionType` of `KEYS_ONLY` or `ALL`.

Partition keys and sort keys

Partition key length

The minimum length of a partition key value is 1 byte. The maximum length is 2048 bytes.

Partition key values

There is no practical limit on the number of distinct partition key values, for tables or for secondary indexes.

Sort key length

The minimum length of a sort key value is 1 byte. The maximum length is 1024 bytes.

Sort key values

In general, there is no practical limit on the number of distinct sort key values per partition key value.

The exception is for tables with secondary indexes. An item collection is the set of items which have the same value of partition key attribute. In a global secondary index the item collection

is independent of the base table (and can have a different partition key attribute), but in a local secondary index the indexed view is colocated in the same partition as the item in the table and shares the same partition key attribute. As a result of this locality, when a table has one or more LSIs, the item collection cannot be distributed to multiple partitions.

For a table with one or more LSIs, item collections cannot exceed 10GB in size. This includes all base table items and all projected LSI views which have the same value of the partition key attribute. 10GB is the maximum size of a partition. For more detailed information, see Item collection size limit.

Naming rules

Table names and Secondary Index names

Names for tables and secondary indexes must be at least 3 characters long, but no greater than 255 characters long. The following are the allowed characters:

- `A-Z`
- `a-z`
- `0-9`
- _ (underscore)
- `-` (hyphen)
- `.` (dot)

Attribute names

In general, an attribute name must be at least one character long, but no greater than 64 KB long.

The following are the exceptions. These attribute names must be no greater than 255 characters long:

- Secondary index partition key names.
- Secondary index sort key names.
- The names of any user-specified projected attributes (applicable only to local secondary indexes). In a `CreateTable` operation, if you specify a `ProjectionType` of `INCLUDE`, the names of the attributes in the `NonKeyAttributes` parameter are length-restricted. The `KEYS_ONLY` and `ALL` projection types are not affected.

These attribute names must be encoded using UTF-8, and the total size of each name (after encoding) cannot exceed 255 bytes.

Data types

String

The length of a String is constrained by the maximum item size of 400 KB.

Strings are Unicode with UTF-8 binary encoding. Because UTF-8 is a variable width encoding, DynamoDB determines the length of a String using its UTF-8 bytes.

Number

A Number can have up to 38 digits of precision, and can be positive, negative, or zero.

- Positive range: `1E-130` to `9.99999999999999999999999999999999999999E+125`
- Negative range: `-9.99999999999999999999999999999999999999E+125` to `-1E-130`

DynamoDB uses JSON strings to represent Number data in requests and replies. For more information, see DynamoDB low-level API.

If number precision is important, you should pass numbers to DynamoDB using strings that you convert from a number type.

Binary

The length of a Binary is constrained by the maximum item size of 400 KB.

Applications that work with Binary attributes must encode the data in base64 format before sending it to DynamoDB. Upon receipt of the data, DynamoDB decodes it into an unsigned byte array and uses that as the length of the attribute.

Items

Item size

The maximum item size in DynamoDB is 400 KB, which includes both attribute name binary length (UTF-8 length) and attribute value lengths (again binary length). The attribute name counts towards the size limit.

For example, consider an item with two attributes: one attribute named "shirt-color" with value "R" and another attribute named "shirt-size" with value "M". The total size of that item is 23 bytes.

Item size for tables with Local Secondary Indexes

For each local secondary index on a table, there is a 400 KB limit on the total of the following:

- The size of an item's data in the table.
- The size of corresponding entries (including key values and projected attributes) in all local secondary indexes.

Attributes

Attribute name-value pairs per item

The cumulative size of attributes per item must fit within the maximum DynamoDB item size (400 KB).

Number of values in list, map, or set

There is no limit on the number of values in a List, a Map, or a Set, as long as the item containing the values fits within the 400 KB item size limit.

Attribute values

Empty String and Binary attribute values are allowed, if the attribute is not used as a key attribute for a table or index. Empty String and Binary values are allowed inside Set, List, and Map types. An attribute value cannot be an an empty Set (String Set, Number Set, or Binary Set). However, empty Lists and Maps are allowed.

Nested attribute depth

DynamoDB supports nested attributes up to 32 levels deep.

Expression parameters

Expression parameters include `ProjectionExpression`, `ConditionExpression`, `UpdateExpression`, and `FilterExpression`.

Lengths

The maximum length of any expression string is 4 KB. For example, the size of the `ConditionExpression a=b` is 3 bytes.

The maximum length of any single expression attribute name or expression attribute value is 255 bytes. For example, `#name` is 5 bytes; `:val` is 4 bytes.

The maximum length of all substitution variables in an expression is 2 MB. This is the sum of the lengths of all `ExpressionAttributeNames` and `ExpressionAttributeValues`.

Operators and operands

The maximum number of operators or functions allowed in an `UpdateExpression` is 300. For example, the *UpdateExpression* `SET a = :val1 + :val2 + :val3` contains two "+" operators.

The maximum number of operands for the `IN` comparator is 100.

Reserved words

DynamoDB does not prevent you from using names that conflict with reserved words. (For a complete list, see Reserved words in DynamoDB.)

However, if you use a reserved word in an expression parameter, you must also specify `ExpressionAttributeNames`. For more information, see Expression attribute names (aliases) in DynamoDB.

DynamoDB transactions

DynamoDB transactional API operations have the following constraints:

- A transaction cannot contain more than 100 unique items.
- A transaction cannot contain more than 4 MB of data.
- No two actions in a transaction can work against the same item in the same table. For example, you cannot both `ConditionCheck` and `Update` the same item in one transaction.
- A transaction cannot operate on tables in more than one AWS account or Region.
- Transactional operations provide atomicity, consistency, isolation, and durability (ACID) guarantees only within the AWS Region where the write is made originally. Transactions are

not supported across Regions in global tables. For example, suppose that you have a global table with replicas in the US East (Ohio) and US West (Oregon) Regions and you perform a `TransactWriteItems` operation in the US East (N. Virginia) Region. In this case, you might observe partially completed transactions in the US West (Oregon) Region as changes are replicated. Changes are replicated to other Regions only after they have been committed in the source Region.

DynamoDB Streams

Simultaneous readers of a shard in DynamoDB Streams

For single-Region tables that are not global tables, you can design for up to two processes to read from the same DynamoDB Streams shard at the same time. Exceeding this limit can result in request throttling. For global tables, we recommend you limit the number of simultaneous readers to one to avoid request throttling.

DynamoDB Accelerator (DAX)

AWS Region availability

For a list of AWS Regions in which DAX is available, see DynamoDB Accelerator (DAX) in the *AWS General Reference*.

Nodes

A DAX cluster consists of exactly one primary node, and between zero and ten read replica nodes.

The total number of nodes (per AWS account) cannot exceed 50 in a single AWS Region.

Parameter groups

You can create up to 20 DAX parameter groups per Region.

Subnet groups

You can create up to 50 DAX subnet groups per Region.

Within a subnet group, you can define up to 20 subnets.

> **Important**
>
> A DAX cluster supports a maximum of 500 DynamoDB tables. Once you go beyond 500 DynamoDB tables, your cluster may experience degradation in availability and performance.

API-specific constraints

`CreateTable/UpdateTable/DeleteTable/PutResourcePolicy/DeleteResourcePolicy`

In general, you can have up to 500 CreateTable, UpdateTable, DeleteTable, PutResourcePolicy, and DeleteResourcePolicy requests running simultaneously in any combination. As a result, the total number of tables in the `CREATING`, `UPDATING`, or `DELETING` state cannot exceed 500.

You can submit up to 2,500 requests per second of mutable (`CreateTable`, `DeleteTable`, `UpdateTable`, `PutResourcePolicy`, and `DeleteResourcePolicy`) control plane API requests across a group of tables. However, the `PutResourcePolicy` and `DeleteResourcePolicy` requests have lower individual limits. For more information, see the following quotas details for `PutResourcePolicy` and `DeleteResourcePolicy`.

`CreateTable` and `PutResourcePolicy` requests which include a resource-based policy will count as two additional requests for each KB of the policy. For example, a `CreateTable` or `PutResourcePolicy` request with a policy of size 5 KB will count as 11 requests. 1 for the `CreateTable` request and 10 for the resource-based policy (2 x 5 KB). Similarly, a policy of size 20 KB will count as 41 requests. 1 for the `CreateTable` request and and 40 for the resource-based policy (2 x 20 KB).

`PutResourcePolicy`

You can submit up to 25 `PutResourcePolicy` API requests per second across a group of tables. After a successful request for an individual table, no new `PutResourcePolicy` requests are supported for the following 15 seconds.

The maximum size supported for a resource-based policy document is 20 KB. DynamoDB counts whitespaces when calculating the size of a policy against this limit.

DeleteResourcePolicy

You can submit up to 50 `DeleteResourcePolicy` API requests per second across a group of tables. After a successful `PutResourcePolicy` request for an individual table, no `DeleteResourcePolicy` requests are supported for the following 15 seconds.

BatchGetItem

A single `BatchGetItem` operation can retrieve a maximum of 100 items. The total size of all the items retrieved cannot exceed 16 MB.

BatchWriteItem

A single `BatchWriteItem` operation can contain up to 25 `PutItem` or `DeleteItem` requests. The total size of all the items written cannot exceed 16 MB.

DescribeStream

You can call `DescribeStream` at a maximum rate of 10 times per second.

DescribeTableReplicaAutoScaling

`DescribeTableReplicaAutoScaling` method supports only 10 requests per second.

DescribeLimits

`DescribeLimits` should be called only periodically. You can expect throttling errors if you call it more than once in a minute.

DescribeContributorInsights/ListContributorInsights/UpdateContributorInsights

`DescribeContributorInsights`, `ListContributorInsights` and `UpdateContributorInsights` should be called only periodically. DynamoDB supports up to five requests per second for each of these APIs.

DescribeTable/ListTables/GetResourcePolicy

You can submit up to 2,500 requests per second of a combination of read-only (`DescribeTable`, `ListTables`, and `GetResourcePolicy`) control plane API requests. The `GetResourcePolicy` API has a lower individual limit of 100 requests per second.

DescribeTimeToLive

The `DescribeTimeToLive` operation is throttled at 10 read request units per second. If you exceed this limit, DynamoDB returns a `ThrottlingException` error.

Query

The result set from a `Query` is limited to 1 MB per call. You can use the `LastEvaluatedKey` from the query response to retrieve more results.

Scan

The result set from a `Scan` is limited to 1 MB per call. You can use the `LastEvaluatedKey` from the scan response to retrieve more results.

UpdateKinesisStreamingDestination

When performing `UpdateKinesisStreamingDestination` operations, you can set `ApproximateCreationDateTimePrecision` to a new value a maximum of 3 times in a 24 hour period.

UpdateTableReplicaAutoScaling

`UpdateTableReplicaAutoScaling` method supports only ten requests per second.

UpdateTableTimeToLive

The `UpdateTableTimeToLive` method supports only one request to enable or disable `Time to Live (TTL)` per specified table per hour. This change can take up to one hour to fully process. Any additional `UpdateTimeToLive` calls for the same table during this one hour duration result in a ValidationException.

DynamoDB encryption at rest

You can switch between an AWS owned key, an AWS managed key, and a customer managed key up to four times, anytime per 24-hour window, on a per table basis, starting from when the table was created. If there was no change in the past six hours, an additional change is allowed. This effectively brings the maximum number of changes in a day to eight (four changes in the first six hours, and one change for each of the subsequent six hour windows in a day).

You can switch encryption keys to use an AWS owned key as often as necessary, even if the above quota has been exhausted.

These are the quotas unless you request a higher amount. To request a service quota increase, see https://aws.amazon.com/support.

DynamoDB API reference

The Amazon DynamoDB API reference contains a complete list of operations supported by:

- DynamoDB
- DynamoDB Streams
- DynamoDB Accelerator (DAX)

Troubleshooting Amazon DynamoDB

The following topics provide troubleshooting advice for errors and issues that you could encounter when using Amazon DynamoDB. If you find an issue that isn't listed here, you can use the feedback button on this page to report it.

For more troubleshooting advice and answers to common support questions, visit the AWS Knowledge Center.

Topics

- Troubleshooting internal server errors in Amazon DynamoDB
- Troubleshooting latency issues in Amazon DynamoDB
- Troubleshooting throttling in Amazon DynamoDB

Troubleshooting internal server errors in Amazon DynamoDB

In DynamoDB, internal server errors (500 errors) indicate that the service is unable to serve the request. These errors can occur for various reasons, such as transient network issues in the fleet, infrastructure issues, storage node related issues, and more.

You may encounter some internal server errors during the lifecycle of your DynamoDB table. This is expected due to the distributed nature of the service and usually shouldn't be a cause for concern. DynamoDB automatically repairs and heals any transient issues with the service in real time, without requiring any intervention from you. However, if you observe a consistently high number of internal server errors on requests to your table (as seen in the the section called “SystemErrors” metric), you should investigate further.

Topics

- Investigating internal server errors
- Minimizing the impact from internal server errors
- Improving operational awareness

Investigating internal server errors

If you encounter internal server errors in your DynamoDB table, consider these options:

1. **Check the AWS Health Dashboard.**

   To identify the issue, the first step is to check the AWS Service Health Dashboard and your AWS Account Health Dashboard. These dashboards provide valuable information about any service-wide issues, impacted tables, ongoing problems, and the root cause once the issue has been resolved.

   Reviewing the details in these dashboards will give you a better understanding of the current status of the AWS services you're using and any potential problems affecting your account. This information can help you determine the next steps to address the issue and minimize any disruptions to your operations.

2. **Reach out to Support.**

   If you observe prolonged, sustained errors in your requests, it may indicate an issue with the service. As a general rule, if you see an overall failure rate of 1% or more over the last 15 minutes, it's an appropriate time to escalate the issue to the AWS Support team. See, DynamoDB Service Level Agreement to learn more.

   When opening a case with the AWS Support team, provide the following details to help expedite the troubleshooting process:

   - Impacted DDB; tables or secondary indexes
   - Time window when the errors were observed
   - DynamoDB request IDs, such as `4KBNVRGD25RG1KE09UT4V3FQDJVV4KQNSO5AEMVJF66Q9ASUAAJG`, which you can find in your application logs.

   Including these details in the support case will help the AWS team understand the problem and provide a faster resolution. If you don't have the request IDs, you should still log the case with the other available details.

Minimizing the impact from internal server errors

If internal server errors happen when using DynamoDB, minimize the impact of these on your application, consider the following best practices:

- Use backoffs and retries – DynamoDB's default SDK behaviors are designed to find the right balance for most applications in terms of back-off and retry strategy. However, you can

adjust these settings based on your application's tolerance for downtime and performance requirements. Learn more about back-offs and retries to understand how you can fine-tune these retry settings.

- Use eventually consistent reads – If your application doesn't require strongly consistent reads, consider using eventually consistent reads. These reads are lower cost and less likely to experience transient issues due to internal server errors as it would be served from any of the available Storage Nodes. For more information, see DynamoDB read consistency.

Improving operational awareness

Maintaining high availability and reliability of your applications is crucial in today's digital landscape. One key aspect of this is proactively monitoring for internal server errors (ISEs) in your DynamoDB tables and global secondary indexes (GSIs). By creating CloudWatch alarms to monitor these errors, you can gain better operational awareness and be alerted to potential issues before they impact your end-users. This approach aligns with the Operational Excellence pillar of the AWS Well-Architected Framework, ensuring your DynamoDB workload is optimized for performance, security, and reliability.

Creating CloudWatch alarms

You should have CloudWatch alarms set on your DynamoDB tables to receive notifications for consistently high numbers of internal server errors instead of observing the metrics manually. This ties with the operational excellence pillar of the Well-Architected framework for any workload on AWS. See Using the DynamoDB Well-Architected Lens to optimize your DynamoDB workload to learn more about Well-Architecting your DynamoDB tables.

These alarms use custom metric math to calculate the failed request percentage for a 5-minute window. The recommended best practice is to configure the alarm to enter the `ALARM` state when 3 consecutive data points breach the 1% threshold, which means that overall 1% of requests fail within a 15-minute period.

The sample below is a CloudFormation template that can help you create CloudWatch alarms on your table and GSI on the table.

```
AWSTemplateFormatVersion: "2010-09-09"
Description: Sample template for monitoring DynamoDB
Parameters:
 DynamoDBProvisionedTableName:
    Description: Name of DynamoDB Provisioned Table to create
```

```
    Type: String
    MinLength: 3
    MaxLength: 255
    ConstraintDescription : https://docs.aws.amazon.com/amazondynamodb/latest/
developerguide/Limits.html#limits-naming-rules
  DynamoDBSNSEmail:
    Description : Email Address subscribed to newly created SNS Topic
    Type: String
    AllowedPattern: "^[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+\\.[a-zA-Z0-9-.]+$"
    MinLength: 1
    MaxLength: 255

Resources:
  DynamoDBMonitoringSNSTopic:
    Type: AWS::SNS::Topic
    Properties:
      DisplayName: DynamoDB Monitoring SNS Topic
      Subscription:
        - Endpoint: !Ref DynamoDBSNSEmail
          Protocol: email
      TopicName: dynamodb-monitoring

  DynamoDBTableSystemErrorAlarm:
    Type: 'AWS::CloudWatch::Alarm'
    Properties:
      AlarmName: 'DynamoDBTableSystemErrorAlarm'
      AlarmDescription: 'Alarm when system errors exceed 1% of total number of requests
 for 15 minutes'
      AlarmActions:
        - !Ref DynamoDBMonitoringSNSTopic
      Metrics:
        - Id: 'e1'
          Expression: 'm1/(m1+m2+m3)'
          Label: SystemErrorsOverTotalRequests
        - Id: 'm1'
          MetricStat:
            Metric:
              Namespace: 'AWS/DynamoDB'
              MetricName: 'SystemErrors'
              Dimensions:
                - Name: 'TableName'
                  Value: !Ref DynamoDBProvisionedTableName
            Period: 300
            Stat: 'SampleCount'
```

```
            Unit: 'Count'
          ReturnData: False
        - Id: 'm2'
          MetricStat:
            Metric:
              Namespace: 'AWS/DynamoDB'
              MetricName: 'ConsumedReadCapacityUnits'
              Dimensions:
                - Name: 'TableName'
                  Value: !Ref DynamoDBProvisionedTableName
            Period: 300
            Stat: 'SampleCount'
            Unit: 'Count'
          ReturnData: False
        - Id: 'm3'
          MetricStat:
            Metric:
              Namespace: 'AWS/DynamoDB'
              MetricName: 'ConsumedWriteCapacityUnits'
              Dimensions:
                - Name: 'TableName'
                  Value: !Ref DynamoDBProvisionedTableName
            Period: 300
            Stat: 'SampleCount'
            Unit: 'Count'
          ReturnData: False
      EvaluationPeriods: 3
      Threshold: 1.0
      ComparisonOperator: 'GreaterThanThreshold'
  DynamoDBGSISystemErrorAlarm:
    Type: 'AWS::CloudWatch::Alarm'
    Properties:
      AlarmName: 'DynamoDBGSISystemErrorAlarm'
      AlarmDescription: 'Alarm when GSI system errors exceed 2% of total number of
requests for 15 minutes'
      AlarmActions:
        - !Ref DynamoDBMonitoringSNSTopic
      Metrics:
        - Id: 'e1'
          Expression: 'm1/(m1+m2+m3)'
          Label: GSISystemErrorsOverTotalRequests
        - Id: 'm1'
          MetricStat:
            Metric:
```

```
            Namespace: 'AWS/DynamoDB'
            MetricName: 'SystemErrors'
            Dimensions:
              - Name: 'TableName'
                Value: !Ref DynamoDBProvisionedTableName
              - Name: 'GlobalSecondaryIndexName'
                Value: !Join [ '-', [!Ref DynamoDBProvisionedTableName, 'gsi1'] ]
          Period: 300
          Stat: 'SampleCount'
          Unit: 'Count'
        ReturnData: False
      - Id: 'm2'
        MetricStat:
          Metric:
            Namespace: 'AWS/DynamoDB'
            MetricName: 'ConsumedReadCapacityUnits'
            Dimensions:
              - Name: 'TableName'
                Value: !Ref DynamoDBProvisionedTableName
              - Name: 'GlobalSecondaryIndexName'
                Value: !Join [ '-', [!Ref DynamoDBProvisionedTableName, 'gsi1'] ]
          Period: 300
          Stat: 'SampleCount'
          Unit: 'Count'
        ReturnData: False
      - Id: 'm3'
        MetricStat:
          Metric:
            Namespace: 'AWS/DynamoDB'
            MetricName: 'ConsumedWriteCapacityUnits'
            Dimensions:
              - Name: 'TableName'
                Value: !Ref DynamoDBProvisionedTableName
              - Name: 'GlobalSecondaryIndexName'
                Value: !Join [ '-', [!Ref DynamoDBProvisionedTableName, 'gsi1'] ]
          Period: 300
          Stat: 'SampleCount'
          Unit: 'Count'
        ReturnData: False
    EvaluationPeriods: 3
    Threshold: 1.0
    ComparisonOperator: 'GreaterThanThreshold'
```

Troubleshooting latency issues in Amazon DynamoDB

If your workload appears to experience high latency, you can analyze the CloudWatch `SuccessfulRequestLatency` metric, and check the average latency and median latency through percentile metrics (p50) to see if it's related to DynamoDB. Some variability in the reported `SuccessfulRequestLatency` is normal, and occasional spikes (particularly in the `Maximum` statistic and high percentiles) should not be cause for concern. However, if the `Average` statistic or p50 (median) shows a sharp increase and persists, you should check the AWS Service Health Dashboard and your Personal Health Dashboard for more information. Some possible causes include the size of the item in your table (a 1 KB item and a 400 KB item will vary in latency) or the size of the query (10 items versus 100 items).

The percentile metrics (p99, p90, etc.) can help you better understand your latency distribution. For example:

- p50 (median) shows the typical latency for your workload.
- p90 shows that 90 percent of requests are faster than this value.
- p99 helps identify the worst-case latency affecting 1 percent of requests.

High p99 values with normal p50 values might indicate sporadic issues affecting a small portion of requests, while consistently elevated p50 values might suggest some performance degradation.

> **Note**
>
> To analyze custom percentile values (such as p99.9), you can manually enter the desired percentile (e.g., p99.9) in the CloudWatch metric statistic field. This allows you to evaluate latency distributions beyond the default percentiles listed in the dropdown.

Some variation in latency metrics, particularly in higher percentiles, is expected and can be seen as a result of DynamoDB-driven background operations that help maintain high availability and durability for your data stored in DynamoDB tables or transient infrastructure issues.

If necessary, consider opening a support case with AWS Support, and continue to assess any available fall-back options for your application (such as evacuation of a Region if you have a multi-Region architecture) according to your runbooks. You should log request IDs for slow requests for providing these IDs to AWS Support when you open a support case.

The `SuccessfulRequestLatency` metric only measures latency which is internal to the DynamoDB service — client side activity and network trip times are not included. To learn more about overall latency for calls from your client to the DynamoDB service, you can enable latency metric logging in your AWS SDK.

> **Note**
>
> For most singleton operations (operations which apply to a single item by fully specifying the primary key's value), DynamoDB delivers single-digit millisecond `Average SuccessfulRequestLatency`. This value does not include the transport overhead for the caller code accessing the DynamoDB endpoint. For multi-item data operations, the latency will vary based on factors such as size of the result set, the complexity of the data structures returned, and any condition expressions and filter expressions applied. For repeated multi-item operations to the same data set with the same parameters, DynamoDB will provide highly consistent `Average SuccessfulRequestLatency`.

Consider one or more of the following strategies to reduce latency:

- **Reuse connections:** DynamoDB requests are made via an authenticated session over HTTPS by default. Initiating the connection requires multiple round-trips and takes time so the latency of the first request is higher than following requests that reuse the connection. Requests over an already initialized connection deliver DynamoDB's consistent low latency. To avoid the latency overhead of establishing new connections, you may want to implement a "keep-alive" mechanism by sending a `GetItem` request every 30 seconds if no other requests are made.
- **Use eventually consistent reads:** If your application doesn't require strongly consistent reads, consider using the default eventually consistent reads. Eventually consistent reads have lower cost and can come from multiple availability zones, allowing selection of an availability zone co-located to the requester which decreases latency. For more information, see DynamoDB read consistency.
- **Implement request hedging:** For very low p99 latency requirements, consider implementing request hedging. With request hedging, if the initial request doesn't receive a response quickly enough, send a second equivalent request and let them race, first response wins. This improves tail latency at the cost of some extra requests. You can decide how long to wait before sending the second request. Hedging is easier for reads. For writes, use timestamp-based ordering to ensure hedged requests are treated as occurring at the time of the first attempt, preventing out-

of-order updates. This approach has been discussed in Timestamp writes for write hedging in Amazon DynamoDB.

- **Adjust request timeout and retry behavior:** The path from your client to DynamoDB traverses many components, each designed with redundancy in mind. Consider the following aspects:
  - Network resiliency
  - TCP packet timeouts
  - DynamoDB's distributed architecture

  Default SDK behaviors are optimized for most applications. However, you can implement a fail-fast strategy and adjust timeout settings. Requests taking significantly longer than normal are less likely to ultimately succeed. By failing fast and retrying, you may quickly succeed through a different path. This is similar to request hedging but ends the first request instead of allowing it to proceed.

  Avoid setting timeout values too low. Overly low timeouts can lead to client-induced availability issues. For example, a 50-millisecond socket timeout could cause connection errors during network latency spikes, such as when approaching Amazon EC2 instance bandwidth limits for single-flow traffic. Carefully weigh the benefits of lower timeouts against potential risks to application availability, and prefer hedging to short timeouts.

  For a helpful discussion on this topic, see Tuning AWS Java SDK HTTP request settings for latency-aware Amazon DynamoDB applications.
- **Reduce the distance between the client and DynamoDB endpoint:** If you have globally dispersed users, consider using Global tables - multi-active, multi-Region replication. With global tables, you can replicate your table to specified AWS Regions where you want the table to be available. You can place a copy of the data closer to the end user to reduce network latency during read and write operations. For more information about using DynamoDB global tables effectively, see Using Amazon DynamoDB global tables in AWS Prescriptive Guidance.
- **Use caching:** If your traffic is read heavy, consider using one of these caching services:
  - DynamoDB Accelerator (DAX): A fully managed, highly available, in-memory cache for DynamoDB that delivers up to a 10x performance improvement, from milliseconds to microseconds, even at millions of requests per second. For more information about DAX, see In-memory acceleration with DynamoDB Accelerator (DAX):
  - Amazon ElastiCache: A fully managed, in-memory caching service that can be integrated with DynamoDB to improve read performance using the cache-aside pattern. For more information,

see Integrating Amazon DynamoDB and Amazon ElastiCache by using read-through caching in AWS Prescriptive Guidance.

Troubleshooting throttling in Amazon DynamoDB

DynamoDB implements throttling for two primary purposes: maintaining overall service performance and cost control. Throttling either serves as an intentional safeguard that prevents performance degradation when consumption rates exceed capacity or as a cost control mechanism when you reach maximum throughput or service quota limits. When throttling occurs, DynamoDB returns specific exceptions with detailed information about why the request was throttled and which resource was impacted. Each reason for throttling corresponds to specific CloudWatch metrics that provide additional insights into the frequency and patterns of throttling events.

The following diagram illustrates the four primary scenarios where DynamoDB implements protective throttling:

1. **Key range throughput exceeded (in both modes)**:

   Consumption directed at specific partitions exceeds the internal partition-level throughput limits.

2. **Provisioned throughput exceeded (in provisioned mode)**:

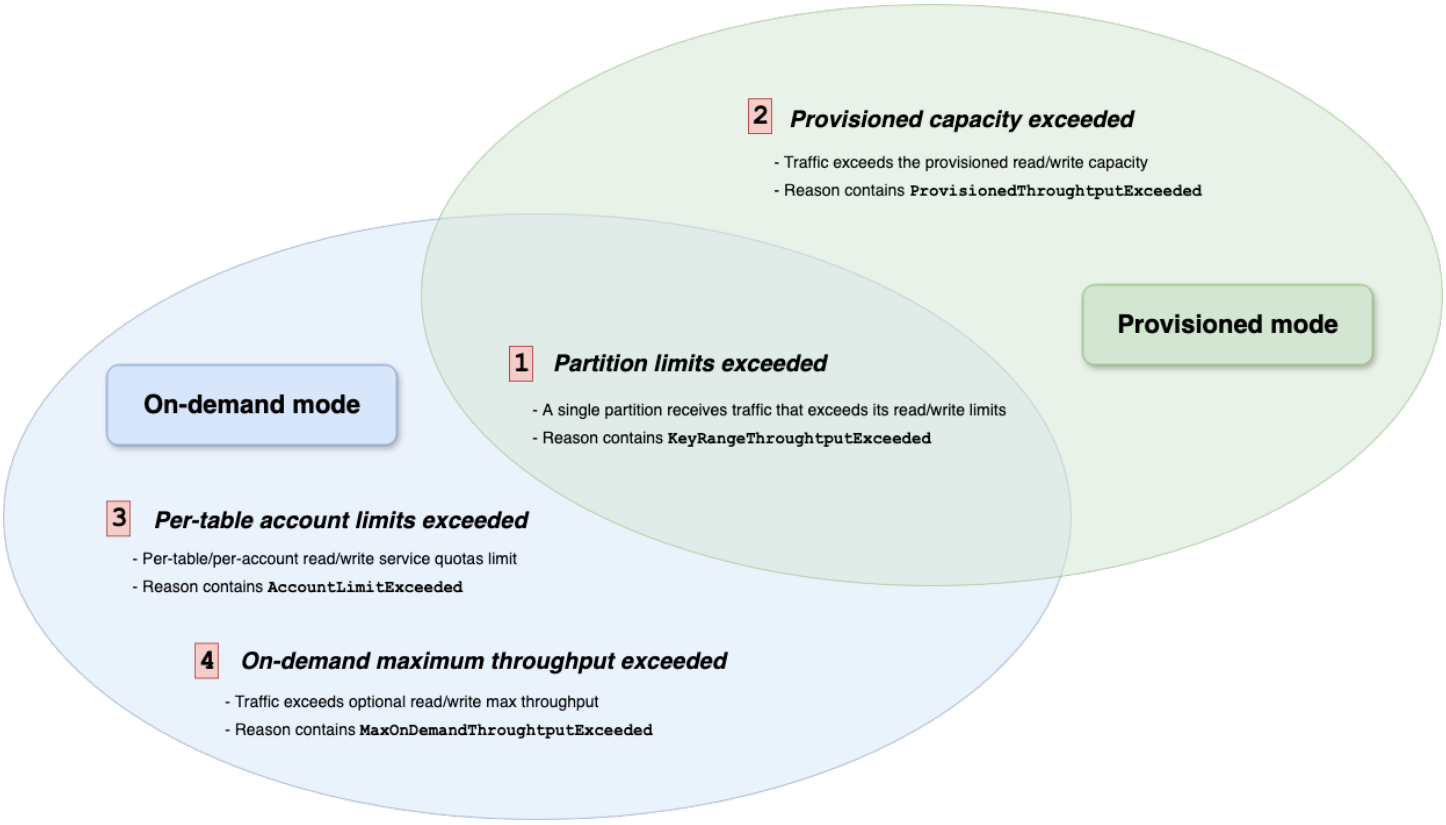
   Consumption exceeds the provisioned capacity units (read or write) configured for a table or a global secondary index (GSI).

3. **Account-level service quotas exceeded (in on-demand mode)**:

   Consumption causes a table or GSI to exceed the per-table account-level service quotas for read/write throughput in the current AWS Region. These quotas serve as backstop safeguards and can be increased.

4. **On-demand maximum throughput exceeded (in on-demand mode)**:

   Consumption exceeds the configured maximum throughput limits set for a table or GSI. These are limits you configure specifically for cost control purposes.



This guide is organized to help you understand and work with throttling in DynamoDB. First, we help you identify the specific type of throttling affecting your workload through a diagnostic framework.

Then, the resolution guide section offers specific guidance for each throttling scenario, including CloudWatch metrics to monitor for detection and analysis, and recommended steps for optimization. By following this structured approach, you can better diagnose the root cause of throttling and implement the appropriate solution to ensure your DynamoDB tables operate efficiently.

To get started, follow the section called “Diagnosing throttling” to learn how to identify which type of throttling is affecting your workload and implement the recommended resolution strategy.

Topics

- Diagnosing throttling
- DynamoDB throttling resolution guide
- Understanding Global Secondary Index (GSI) write throttling and back pressure in DynamoDB
- CloudWatch throttling metrics

Diagnosing throttling

When your application experiences throttling, DynamoDB provides detailed exception information and targeted CloudWatch metrics to help you diagnose these events.

This section presents a systematic approach to understanding throttling events in your DynamoDB applications. It shows how to interpret throttling exceptions, correlate them with CloudWatch metrics for deeper insights, and understand what changes would reduce throttling in your DynamoDB applications.

Understanding throttling exceptions

When DynamoDB throttles a request, it returns specific exceptions with detailed diagnostic information. For example, in Java, these include `ProvisionedThroughputExceededException`, `RequestLimitExceeded`, or `ThrottlingException`.

Each exception includes `ThrottlingReasons`, a collection of individual `ThrottlingReason` containing two key fields to help you identify and understand the throttling:

- **A reason** - A concatenated field following the format `<ResourceType><OperationType><LimitType>`
- **A resource ARN** - The Amazon Resource Name (ARN) of the affected table or index

The reason field follows a consistent patern that helps you understand exactly what's happening:

- **ResourceType** (What is being throttled): `Table` or `Index`
- **OperationType** (What kind of operation): `Read` or `Write`
- **LimitType** (Why the throttling occurred):
  - `KeyRangeThroughputExceeded`: This occurs when a specific partition backing your table or index has consumed read or write capacity exceeding the internal per-partition throughput limits.
  - `ProvisionedThroughputExceeded`: This happens on a provisioned table or global secondary index when the read or write consumption rate has exceeded the provisioned amount.
  - `AccountLimitExceeded`: This happens on an on-demand table or index when the read or write consumption rate has exceeded the maximum consumption rate for a table and its indexes as set at the account level. You can request a raise in this quota.

- `MaxOnDemandThroughputExceeded`: This happens on an on-demand table or index when the read or write consumption rate has exceeded the user-provided maximum consumption rate configured for the table or index. You can raise this value yourself to any value up to the account limit or set to -1 to indicate no user-provided limit.

The resource ARN identifies exactly which table or index is being throttled:

- For tables: `arn:aws:dynamodb:[region]:[account-id]:table/[table-name]`
- For indexes: `arn:aws:dynamodb:[region]:[account-id]:table/[table-name]/index/[index-name]`

Examples of complete throttling reasons:

- `TableReadProvisionedThroughputExceeded`
- `IndexWriteAccountLimitExceeded`

This helps identify exactly what resource is being throttled, what type of operation caused it, and why the throttling occurred.

Example exceptions

Example 1: Provisioned capacity exceeded on a GSI

```
{
    "ThrottlingReasons": [
        {
            "reason": "IndexWriteProvisionedThroughputExceeded",
            "resource": "arn:aws:dynamodb:us-west-2:123456789012:table/CustomerOrders/
index/OrderDateIndex"
        }
    ],
    "awsErrorDetails": {
        "errorCode": "ProvisionedThroughputExceeded",
        "errorMessage": "The level of configured provisioned throughput for the index
 was exceeded",
        "serviceName": "DynamoDB",
        "sdkHttpResponse": {
            "statusText": "Bad Request",
            "statusCode": 400
```

```
        }
    }
}
```

In this example, the application receives a `ProvisionedThroughputExceededException` with the reason `IndexWriteProvisionedThroughputExceeded`. Writes to the `OrderDateIndex` are being throttled because the write consumption has exceeded the GSI's configured provisioned write capacity.

Example 2: On-demand maximum throughput exceeded

```
{
    "ThrottlingReasons": [
        {
            "reason": "TableReadMaxOnDemandThroughputExceeded",
            "resource": "arn:aws:dynamodb:us-east-1:123456789012:table/UserSessions"
        }
    ],
    "awsErrorDetails": {
        "errorMessage": "Throughput exceeds the maximum OnDemandThroughput configured
 on table or index",
        "errorCode": "ThrottlingException",
        "serviceName": "DynamoDB",
        "sdkHttpResponse": {
            "statusText": "Bad Request",
            "statusCode": 400
        }
    }
}
```

In this example, reads from the `UserSessions` table are being throttled because they exceed the configured maximum on-demand throughput limit on the table.

DynamoDB throttling diagnosis framework

When your application encounters throttling, follow these steps to diagnose and resolve the issue.

Step 1 - Analyze the `ThrottlingReason` details

1. Check the **reason** field to identify the specific reason for throttling. The reason details the type of resource throttled (table or index), the type of operation causing the throttling event (read or write), and the limit type that was exceeded (partition, provisioned throughput, account limit).

2. Check the **resourceArn** field to identify which resource (table or GSI) is being throttled.
3. Use this combined information to understand the full context of the throttling issue.

   For example, consider this scenario where you receive the following exception `ProvisionedThroughputExceededException` with a throttling reason `TableWriteKeyRangeThroughputExceeded`. The impacted resourceARN is `arn:aws:dynamodb:us-west-2:123456789012:table/CustomerOrders`.

   This combination informs you that write operations to your `CustomerOrders` table are being throttled. The throttling is occurring at the partition level (not the table level, which would show as `TableWriteProvisionedThroughputExceeded`). The root cause is that you've exceeded the maximum throughput capacity for a specific partition key value or range, indicating a hot partition issue.

   Understanding this relationship between the exception elements helps you implement the appropriate mitigation strategy - in this case, addressing the hot partition rather than increasing the table's overall provisioned capacity.

Step 2 - Identify and analyze the related CloudWatch metrics

1. **Identify your metrics:** Each throttling reason in DynamoDB directly corresponds to specific CloudWatch metrics that you can monitor to track and analyze throttling events. You can systematically derive the appropriate CloudWatch metric names from the throttling reason.
2. Match your throttling reason to the corresponding CloudWatch metrics using this reference table:

   Complete throttling reasons and CloudWatch metrics reference

Category	Throttling reason	Primary CloudWatch metrics
Provisioned capacity exceeded	TableReadProvisionedThroughputExceeded	ReadProvisionedThroughputThrottleEvents
	TableWriteProvisionedThroughputExceeded	WriteProvisionedThroughputThrottleEvents

Category	Throttling reason	Primary CloudWatch metrics
	IndexReadProvisionedThroughputExceeded	ReadProvisionedThroughputThrottleEvents (GSI)
	IndexWriteProvisionedThroughputExceeded	WriteProvisionedThroughputThrottleEvents (GSI)
Partition limits exceeded	TableReadKeyRangeThroughputExceeded	ReadKeyRangeThroughputThrottleEvents
	TableWriteKeyRangeThroughputExceeded	WriteKeyRangeThroughputThrottleEvents
	IndexReadKeyRangeThroughputExceeded	ReadKeyRangeThroughputThrottleEvents (GSI)
	IndexWriteKeyRangeThroughputExceeded	WriteKeyRangeThroughputThrottleEvents (GSI)
On-demand maximum exceeded	TableReadMaxOnDemandThroughputExceeded	ReadMaxOnDemandThroughputThrottleEvents
	TableWriteMaxOnDemandThroughputExceeded	WriteMaxOnDemandThroughputThrottleEvents
	IndexReadMaxOnDemandThroughputExceeded	ReadMaxOnDemandThroughputThrottleEvents (GSI)

Category	Throttling reason	Primary CloudWatch metrics
	IndexWriteMaxOnDemandThroughputExceeded	WriteMaxOnDemandThroughputThrottleEvents (GSI)
Account limits exceeded	TableReadAccountLimitExceeded	ReadAccountLimitThrottleEvents
	TableWriteAccountLimitExceeded	WriteAccountLimitThrottleEvents
	IndexReadAccountLimitExceeded	ReadAccountLimitThrottleEvents (GSIs)
	IndexWriteAccountLimitExceeded	WriteAccountLimitThrottleEvents (GSIs)

For example, if you received `IndexWriteProvisionedThroughputExceeded`, at a minimum, you should monitor the `WriteProvisionedThroughputThrottleEvents` CloudWatch metric for the specific index identified in the `ResourceArn`.

3. Monitor these metrics in CloudWatch to understand the frequency and timing of the throttling events, differentiate between read and write throttling, identify time patterns when throttling increases, and track your capacity utilization trends.

   DynamoDB publishes detailed metrics for each table and global secondary index. The metrics (`ReadThrottleEvents`, `WriteThrottleEvents`, and `ThrottledRequests`) aggregate all throttling events across your table and its indexes.

Step 3 - Identify your throttled keys and high access rates using CloudWatch Contributor Insights (for partition-related throttling)

If you identified partition-related issues in Step 1 (such as `KeyRangeThroughputExceeded` errors), CloudWatch Contributor Insights for DynamoDB can help you diagnose which specific keys are driving your traffic and experiencing throttling events in your table or index.

1. Enable CloudWatch Contributor Insights for your throttled table or index based on your `ResourceARN`.

   You can choose the *Throttled keys* mode to focus exclusively on the most throttled keys. This mode is ideal for continuous monitoring as it only processes events when throttling occurs. Alternatively, the *Accessed and throttled keys* mode helps you look for patterns in your most accessed keys.
2. Analyze the reports to identify problematic patterns. Look for keys with disproportionately high access or throttling rates, correlate throttling and traffic patterns. You can create integrated dashboards combining Contributor Insights graphs and DynamoDB CloudWatch metrics.

For detailed information about enabling and using CloudWatch Contributor Insights, see Using CloudWatch Contributor Insights for DynamoDB.

Step 4 - Determine the appropriate solution

After diagnosing the specific cause of throttling, implement recommended solution based on your specific context. The appropriate solution depends on multiple factors, including your throttling scenario, table's capacity mode, table and key design decisions, access patterns and query efficiency, global and secondary index configuration, and overall system architecture and integration points.

For detailed solutions to address your specific throttling scenarios, see the the section called "Resolution guide" section. This resource provides targeted remediation strategies customized to your particular throttling reason and capacity mode configuration.

Step 5 - Monitor your progress

1. Track your CloudWatch metrics that correspond to your throttling scenario.
2. Validate that your mitigation strategies are effective by observing a decrease in throttling events.

DynamoDB throttling resolution guide

This section provides targeted resolution guidance for each specific throttling reason that DynamoDB may return. Each entry includes suggested resolution approaches based on best practices and corresponding CloudWatch metrics to monitor.

DynamoDB implements 16 distinct throttling reasons across four main categories. Use the throttling reasons from your application's exception to navigate directly to the relevant guidance.

Key range throughput exceeded (hot partitions)

These throttling reasons occur when individual partitions exceed their throughput limits, affecting both provisioned and on-demand modes:

- TableReadKeyRangeThroughputExceeded
- TableWriteKeyRangeThroughputExceeded
- IndexReadKeyRangeThroughputExceeded
- IndexWriteKeyRangeThroughputExceeded

Provisioned throughput exceeded

These throttling reasons occur when consumption rates exceed provisioned capacity limits in provisioned mode:

- TableReadProvisionedThroughputExceeded
- TableWriteProvisionedThroughputExceeded
- IndexReadProvisionedThroughputExceeded
- IndexWriteProvisionedThroughputExceeded

Account limits exceeded

These throttling reasons occur when consumption rates exceed account-level throughput quotas in your AWS Region:

- TableReadAccountLimitExceeded
- TableWriteAccountLimitExceeded
- IndexReadAccountLimitExceeded
- IndexWriteAccountLimitExceeded

On-demand maximum throughput exceeded

These throttling reasons occur when consumption rates exceed configured maximum throughput limits in on-demand mode:

- TableReadMaxOnDemandThroughputExceeded
- TableWriteMaxOnDemandThroughputExceeded
- IndexReadMaxOnDemandThroughputExceeded
- IndexWriteMaxOnDemandThroughputExceeded

1- Key range throughput exceeded (hot partitions)

Amazon DynamoDB enforces specific throughput limits at the partition level for both table and global secondary index (GSI). Each partition has a maximum number of read capacity units (RCUs) and write capacity units (WCUs) per second. When partitions receive concentrated traffic that exceeds these limits, they experience throttling while other operations may remain underutilized, creating "hot partitions." DynamoDB's partition-level throttling operates independently for reads and writes - a partition may throttle reads while writes continue normally, or vice versa. This throttling can occur even when your table or GSI has sufficient overall capacity. To learn more about:

- DynamoDB partition limits and effective partition key design addressing hot partition prevention, see Best practices for designing and using partition keys effectively in DynamoDB.
- General partition concepts and data distribution, see Partitions in DynamoDB.
- Additional guidance and real-world scenarios for managing partition keys and throughput, see the section called “Additional resources”.

When individual partitions exceed their throughput limits, DynamoDB returns a `KeyRangeThroughputExceeded` throttling reason type in the throttling exception. The information identifies that a partition is experiencing high traffic and which operation type (read or write) is causing the issue.

Key range throughput exceeded mitigation measures

This section provides resolution guidance for partition-level throttling scenarios. Before using this guide, ensure you have identified the specific throttling reasons from your application's exception handling, and determined the Amazon Resource Name (ARN) of the affected resource.

For information on retrieving throttling reasons and identifying throttled resources, see the section called “Diagnosis framework”.

Before diving into specific throttling scenarios, first, check if the problem resolves automatically:

- DynamoDB often adapts to hot partitions through its automatic split-for-heat mechanism. If you see throttling events that stop after a short period, your table may have already adapted by splitting the hot partition. When partitions split, each new partition handles a smaller section of the keyspace, which can help distribute the load more evenly. In many cases, no further action is needed as DynamoDB has automatically resolved the issue.

  For more information about the *split-for-heat mechanism*, see the section called “Additional resources”.

If the throttling persists, refer to the specific throttling scenarios below for targeted remediation options:

- the section called “TableReadKeyRangeThroughputExceeded”
- the section called “TableWriteKeyRangeThroughputExceeded”
- the section called “IndexReadKeyRangeThroughputExceeded”
- the section called “IndexWriteKeyRangeThroughputExceeded”

TableReadKeyRangeThroughputExceeded

When this occurs

The consumption rate of one or more partitions in your DynamoDB table exceeds the partition's read throughput limit. This throttling occurs regardless of your table's total provisioned capacity and affects both provisioned and on-demand tables. You can monitor the CloudWatch metrics in the section called “Common diagnosis and monitoring” to analyze your throttling event.

Remediation options

Consider these steps to address your throttling events:

For both provisioned and on-demand modes:

- **Pre-warm capacity:** If throttling persists, check if your table is limited by its the section called “Warm throughput” capacity. Use warm throughput or increase read provisioned capacity in

advance for expected traffic increases. Increasing warm throughput improves your table's ability to handle sudden traffic spikes before throttling occurs. Over time, if your actual throughput consistently approaches the warm throughput levels, DynamoDB may split busy partitions based on observed usage patterns.

- **Identify your hot keys:** If the table didn't resolve it automatically and your warm throughput is high or raising it didn't help, you'll need to identify specific hot keys. Use the section called "Identifying hot keys using CloudWatch Contributor Insights" to determine if any particular partition key values are hot. This is a first step to target your mitigation efforts effectively. Note that identification may not always be straightforward, particularly with rolling hot partitions (where different partitions become hot over time) or when throttling is triggered by operations like scans. For these complex scenarios, you may need to analyze your application's access patterns and correlate them with the timing of throttling events.
- **Depending on your use case, consider using eventually consistent reads:** Switch from strongly consistent to eventually consistent reads, which consume half the RCUs and can immediately double your effective read capacity. For best practices on implementing eventually consistent reads to reduce read capacity consumption, see the section called "DynamoDB read consistency".
- **Improve partition key design:** As a long-term solution, consider the section called "Improving partition key design" to distribute access more evenly across partitions. This approach often provides the most comprehensive resolution to hot partition issues by addressing the root cause. However, it requires careful planning as it involves significant migration challenges.

TableWriteKeyRangeThroughputExceeded

When this occurs

The consumption rate of one or more partitions in your DynamoDB table exceeds the partition's write throughput limit. This throttling occurs regardless of your table's total provisioned capacity and affects both provisioned and on-demand tables. You can monitor the CloudWatch metrics in the section called "Common diagnosis and monitoring" to analyze your throttling event.

Remediation options

Consider these steps to address your throttling events:

For both provisioned and on-demand modes:

- **Pre-warm capacity:** If throttling persists, check if your table is limited by its Understanding DynamoDB warm throughput capacity. Use warm throughput or increase write provisioned

capacity in advance for expected traffic increases. Increasing warm throughput improves your table's ability to handle sudden traffic spikes before throttling occurs. Over time, if your actual throughput consistently approaches the warm throughput levels, DynamoDB may split busy partitions based on observed usage patterns.

- **Identify your hot keys:** If the table didn't resolve it automatically and your warm throughput is high or raising it didn't help, you'll need to identify specific hot keys. Use the section called "Identifying hot keys using CloudWatch Contributor Insights" to determine if any particular partition key values are hot. This is a first step to target your mitigation efforts effectively. Consider these common patterns:
  - If you see the same partition key appearing frequently in your throttling data, this indicates a concentrated hot key.
  - If you do not see repeated keys but are writing data in an ordered way (such as sequential timestamps or scan-based operations that follow keyspace order), you likely have rolling hot partitions where different keys become hot over time as your writes move through the keyspace.

  Note that write throttling can also occur with operations like `BatchWriteItem` or transactions that affect multiple items simultaneously. When individual items within a `BatchWriteItem` request are throttled, DynamoDB does not propagate these throttling errors to the application code. Instead, DynamoDB returns information about the unprocessed items in the response, which your application must handle by retrying those specific items. For transactions, the entire operation fails with a `TransactionCanceledException` if any item experiences throttling. For these complex scenarios, you may need to analyze your application's write patterns and data ingestion workflows, correlate them with the timing of throttling events, and implement appropriate retry handling strategies.
- **Improve partition key design:** As a long-term solution, consider the section called "Improving partition key design" to distribute access more evenly across partitions. This approach often provides the most comprehensive resolution to hot partition issues by addressing the root cause. However, it requires careful planning as it involves significant migration challenges.

IndexReadKeyRangeThroughputExceeded

When this occurs

The consumption rate of one or more partitions in your DynamoDB GSI exceeds the partition's read throughput limit. This throttling occurs regardless of your GSI's total provisioned capacity and

affects both provisioned and on-demand tables. You can monitor the CloudWatch metrics in the section called "Common diagnosis and monitoring" to analyze your throttling event.

Remediation options

Consider these steps to address your throttling events:

- **Pre-warm capacity:** If throttling persists, check if your GSI is limited by its Understanding DynamoDB warm throughput capacity. Use warm throughput or increase read provisioned capacity in advance for expected traffic increases. Increasing warm throughput improves your GSI's ability to handle sudden traffic spikes before throttling occurs. Over time, if your actual throughput consistently approaches the warm throughput levels, DynamoDB may split busy partitions based on observed usage patterns.
- **Identify your hot keys:** If the GSI didn't resolve it automatically and your warm throughput is high or raising it didn't help, you'll need to identify specific hot keys. Use the section called "Identifying hot keys using CloudWatch Contributor Insights" to determine if any particular partition key values are hot. This is a first step to target your mitigation efforts effectively. Note that for GSIs, the partition key distribution may differ significantly from your base table, creating different hot key patterns.
- **Redesign GSI partition keys:** Consider whether your GSI key design might be creating artificial hot spots (such as status flags, date-only keys, or boolean attributes) that concentrate reads on a small number of partitions. Consider using composite keys that combine the low-cardinality attribute with a high-cardinality attribute (e.g., "ACTIVE#customer123" instead of just "ACTIVE") or apply the section called "Write sharding" techniques to the base table items that affect GSI distribution to distribute writes across multiple partitions. While querying sharded data requires additional application logic to aggregate results, this approach prevents throttling by distributing access patterns more evenly.

IndexWriteKeyRangeThroughputExceeded

When this occurs

The consumption rate of one or more partitions in your DynamoDB GSI exceeds the partition's write throughput limit. This throttling occurs regardless of your GSI's total provisioned capacity and affects both provisioned and on-demand tables. You can monitor the CloudWatch metrics in the section called "Common diagnosis and monitoring" to analyze your throttling event.

Remediation options

Consider these steps to address your throttling events:

- **Redesign GSI partition key:** Review your GSI partition key design to verify it has sufficient cardinality (uniqueness) to distribute writes evenly. A common cause of GSI write throttling is using low-cardinality attributes as GSI partition keys (such as status flags with only a few possible values). Even when your base table has well-distributed partition keys, your GSI can still experience hot partitions if its partition key concentrates writes to a small number of values. For example, if 80% of your items have status="ACTIVE", this creates a severe hot partition in a status-based GSI. Consider using composite keys that combine the low-cardinality attribute with a high-cardinality attribute (e.g., "ACTIVE#customer123" instead of just "ACTIVE") or apply the section called “Write sharding” techniques to the base table items that affect GSI distribution to distribute writes across multiple partitions. While querying sharded data requires additional application logic to aggregate results, this approach prevents throttling by distributing access patterns more evenly.
- **Pre-warm capacity:** Check if your GSI is limited by its Understanding DynamoDB warm throughput capacity. Use warm throughput or increase write provisioned capacity in advance for expected traffic increases. Increasing warm throughput improves your GSI's ability to handle sudden traffic spikes before throttling occurs. Over time, if your actual throughput consistently approaches the warm throughput levels, DynamoDB may split busy partitions based on observed usage patterns.
- **Optimize GSI projections:** Apply the section called “Optimizing GSI projections” techniques to reduce write volume to GSIs. Projecting fewer attributes can significantly reduce the write capacity consumed by each GSI update.

Common diagnosis and monitoring

When troubleshooting partition-level throttling, several CloudWatch metrics can help identify hot partitions and confirm the root cause.

Essential CloudWatch metrics

Monitor these key metrics to diagnose partition-level throttling:

- **Partition-level throttling events:** `ReadKeyRangeThroughputThrottleEvents` and `WriteKeyRangeThroughputThrottleEvents` track when individual partitions exceed their throughput limits. `ReadThrottleEvents` and `WriteThrottleEvents` track when any read or write requests exceed the provisioned capacity.

- **Capacity consumption:** `ConsumedReadCapacityUnits` and `ConsumedWriteCapacityUnits` show overall usage patterns.

Resolution procedures

Identifying hot keys using CloudWatch Contributor Insights

Use this procedure to identify which partition keys are causing throttling.

1. Enable CloudWatch Contributor Insights on your table or GSI to track the most throttled keys. Consider keeping CloudWatch Contributor Insights enabled continuously for real-time throttling alerts by using the *Throttled keys* mode. This mode focuses exclusively on throttled requests by only processing events when throttling occurs. This targeted monitoring is a cost effective way to maintain continuous visibility into throttling issues.
2. Identify which keys are causing the hot partition issues.
3. (If the full *Accessed and throttled keys mode* is enabled) Analyze the access patterns over time to determine if hot keys are consistent or occur during specific periods.

Improving partition key design

Use this approach when you can modify your table schema to better distribute traffic across partitions. When possible, this is the most effective long-term solution for hot partition issues. Ideally, partition key design should be carefully considered during the initial table design phase.

Partition key redesign represents a fundamental change to your data model that impacts your entire application ecosystem. Before proceeding with this approach, carefully consider these significant limitations:

- **Data migration complexity:** Redesigning partition keys requires migrating all existing data, which can be resource-intensive and time-consuming for large tables.
- **Application code changes:** All application code that reads or writes to the table must be updated to use the new key structure.
- **Production impact:** Migrating to a new key design often requires downtime or complex dual-write strategies during transition.

For comprehensive guidance and principles on partition key design, see the section called "Partition key design" and the section called "Distributing workloads".

Optimizing GSI projections

Review your application's query patterns to determine exactly which attributes need to be available when querying the GSI, and limit your projections to just those attributes. When you update attributes that aren't projected into a GSI, no write operation occurs on that GSI, reducing write throughput consumption during updates. This targeted projection strategy optimizes both performance and cost while still supporting your application's query requirements. Note that projecting fewer attributes reduces write capacity consumption but may require additional base table reads.

For more information about efficient projection strategies, see Best Practices for Using Secondary Indexes in DynamoDB.

Additional resources

The following blog posts provide hands-on examples and practical details for the concepts covered in this guide:

- For hands-on guidance about scaling DynamoDB and managing hot partitions, see Part 1: Scaling DynamoDB - How partitions, hot keys, and split for heat impact performance.
- For detailed information about how DynamoDB's split-for-heat mechanism works, its benefits, and implementation details, see Part 3: Summary and best practices.
- For detailed write sharding strategies, see the section called “Write sharding”.

2- Provisioned throughput exceeded

Provisioned capacity throttling occurs when your application's consumption rate exceeds the read or write capacity units (RCUs/WCUs) configured for your tables or global secondary indexes. While DynamoDB provides burst capacity to handle occasional traffic spikes, sustained requests beyond your provisioned limits result in throttling. When this happens, DynamoDB returns a `ProvisionedThroughputExceeded` throttling reason type in the throttling exception. The reason identifies whether the issue is with read or write operations and whether it affects the base table or a global secondary index.

Throttling can occur regardless of whether Auto Scaling is enabled. Auto Scaling adapts to increases in consumption, but it does not respond instantly and it is constrained by the maximum capacity limits you configure. This means throttling can still occur during sudden traffic spikes or when consumption exceeds your maximum Auto Scaling limits.

Provisioned throughput exceeded mitigation measures

This section provides resolution guidance for provisioned capacity throttling scenarios. Before using this guide, ensure you have identified the specific throttling reason from your application's exception handling, and determined the Amazon Resource Name (ARN) of the affected resource. For information on retrieving throttling reasons and identifying throttled resources, see the section called “Diagnosis framework”.

Before diving into specific throttling scenarios, first consider if the throttling is actually a problem that needs resolution:

- Occasional throttling is normal and expected in well-optimized DynamoDB applications. Throttling simply means you're consuming 100% of what you've provisioned. If your application handles throttling gracefully with retries and your overall performance meets requirements, the throttling may not require immediate action.
- However, if throttling is causing unacceptable client-side latency, degrading user experience, or preventing critical operations from completing in a timely manner, then proceed with the mitigation options below.

When you need to address throttling issues, first determine if your throttling is caused by:

- **Temporary traffic spikes:** Short-duration increases in traffic that exceed your provisioned capacity but aren't sustained. These require different strategies than continuous high traffic.
- **Continuous high traffic:** Sustained workloads that consistently exceed your provisioned capacity.

For traffic spikes, consider the strategies from the *Handle traffic spikes with Amazon DynamoDB provisioned capacity* blog in the section called “Additional resources”.

For continuous high traffic, consider the capacity adjustment options below:

- the section called “TableReadProvisionedThroughputExceeded”
- the section called “TableWriteProvisionedThroughputExceeded”
- the section called “IndexReadProvisionedThroughputExceeded”
- the section called “IndexWriteProvisionedThroughputExceeded”

TableReadProvisionedThroughputExceeded

When this occurs

Your application's read consumption rate exceeds the provisioned read capacity units (RCUs) configured for your table. You can monitor the CloudWatch metrics in the section called "Common diagnosis and monitoring" to analyze your throttling event.

Resolution approach

Consider these strategies to resolve read capacity throttling:

- **Switch to on-demand capacity mode:** Consider switching your table to on-demand if you experience frequent throttling from traffic spikes. On-demand eliminates provisioning concerns and automatically scales with your workload.
- **If staying with provisioned mode and Auto Scaling is not enabled:**
  - Consider increasing the table read capacity.
  - Enable Auto Scaling for read capacity on your table.
- **If Auto Scaling is enabled (default for tables created in the console):**
  - Optimize your table's read Auto Scaling parameters.

TableWriteProvisionedThroughputExceeded

When this occurs

Your application's write consumption rate exceeds the provisioned write capacity units (WCUs) configured for your table. You can monitor the CloudWatch metrics in the section called "Common diagnosis and monitoring" to analyze your throttling event.

Resolution approach

Consider these strategies to resolve write capacity throttling:

- **Switch to on-demand capacity mode:** Consider switching your table to on-demand if you experience frequent throttling from traffic spikes. On-demand eliminates provisioning concerns and automatically scales with your workload.
- **If staying with provisioned mode and Auto Scaling is not enabled:**
  - Consider increasing the table write capacity.

- Enable Auto Scaling for write capacity on your table.
- **If Auto Scaling is enabled (default for tables created in the console):**
  - Optimize your table's write Auto Scaling parameters.

IndexReadProvisionedThroughputExceeded

When this occurs

Read consumption on a Global Secondary Index (GSI) exceeds the GSI's provisioned read capacity units (RCUs). You can monitor the CloudWatch metrics in the section called "Common diagnosis and monitoring" to analyze your throttling event.

Resolution approach

Consider these strategies to resolve GSI read capacity throttling:

- **Switch to on-demand capacity mode:** Consider switching the base table to on-demand if you experience frequent throttling from traffic spikes. On-demand eliminates provisioning concerns and automatically scales with your workload.
- **If staying with provisioned mode and Auto Scaling is not enabled:**
  - Consider increasing the GSI read capacity.
  - Enable Auto Scaling for read capacity on your GSI.
- **If Auto Scaling is enabled (default for tables created in the console):**
  - Optimize your GSI's read Auto Scaling parameters.

IndexWriteProvisionedThroughputExceeded

When this occurs

Updates to items in the base table trigger writes to a GSI that exceed the GSI's provisioned write capacity. This causes back-pressure throttling on base table writes. You can monitor the CloudWatch metrics in the section called "Common diagnosis and monitoring" to analyze your throttling event.

Resolution approach

Consider these strategies to resolve GSI write capacity throttling:

- **Switch to on-demand capacity mode:** Consider switching the base table to on-demand if you experience frequent throttling from traffic spikes. On-demand eliminates provisioning concerns and automatically scales with your workload.
- **If staying with provisioned mode and Auto Scaling is not enabled:**
  - Consider increasing the GSI write capacity.
  - Enable Auto Scaling for write capacity on your GSI.
- **If Auto Scaling is enabled (default for tables created in the console):**
  - Optimize your GSI's write Auto Scaling parameters.

Common diagnosis and monitoring

When troubleshooting throughput errors, several CloudWatch metrics can help identify the root cause.

Essential CloudWatch metrics

Monitor these key metrics to diagnose provisioned capacity throttling:

- **Throttling events:** `ReadProvisionedThroughputThrottleEvents` and `WriteProvisionedThroughputThrottleEvents` track when requests are throttled for this reason.`ReadThrottleEvents` and `WriteThrottleEvents` track when any read or write requests exceed the provisioned capacity.
- **Capacity consumption:** `ConsumedReadCapacityUnits` and `ConsumedWriteCapacityUnits` show actual usage.
- **Provisioned capacity:** `ProvisionedReadCapacityUnits` and `ProvisionedWriteCapacityUnits` show configured limits.

Resolution procedures

Increasing table throughput capacity

Use this procedure when Auto Scaling is not enabled and you need immediate capacity increase.

1. Update your table's provisioned capacity using the DynamoDB console, AWS CLI, or SDK:
   - **For read capacity:** Increase the `ReadCapacityUnits` parameter, which specifies the maximum number of strongly consistent reads consumed per second before DynamoDB throttles requests.

- **For write capacity:** Increase the `WriteCapacityUnits` parameter, which specifies the maximum number of writes consumed per second before DynamoDB throttles requests.

2. Verify that your new capacity settings don't exceed the per-table throughput quotas and that your total account consumption remains below the per-account throughput quotas for your Region. If you're approaching these limits, consider switching to on-demand capacity mode instead.

Configuring table Auto Scaling to adjust the read or write capacity of your table or GSI

Configure DynamoDB Auto Scaling to automatically adjust read or write capacity based on traffic patterns. You can configure Auto Scaling independently for both tables and GSIs, with separate controls for read and write capacity units.

1. Enable Auto Scaling for read capacity, write capacity, or both on your table or GSI.
2. Set a target utilization percentage with headroom for traffic spikes.

> **Note**
>
> Lower target utilization increases costs and scaling frequency. Targets below 40% may cause over-provisioning. Monitor usage patterns and costs to balance performance and efficiency.

3. Set capacity boundaries:

- **Minimum RCUs/WCUs:** Maintains sufficient capacity during low-traffic periods.
- **Maximum RCUs/WCUs:** Accommodates peak traffic demands and protects against runaway scaling events.

For guidance on configuring and managing DynamoDB Auto Scaling, see Managing throughput capacity automatically with DynamoDB Auto Scaling.

> **Note**
>
> Auto Scaling typically takes several minutes to respond to traffic changes. For sudden traffic spikes, your table's burst capacity provides immediate protection while Auto Scaling

adjusts. Configure target utilization with adequate headroom to allow time for scaling operations and to preserve burst capacity for unexpected demand.

Optimizing your table's or index's read or write Auto Scaling settings

Use this procedure when Auto Scaling is enabled but throttling still occurs. You can tune Auto Scaling independently for both tables and global secondary indexes (GSIs), with separate controls for read and write capacity units.

- **Adjust target utilization:** Consider lowering the target utilization for your table or GSIs to trigger scaling earlier before throttling occurs. Ensure that you monitor your traffic after making these adjustments. See the section called "Configuring table Auto Scaling to adjust the read or write capacity of your table or GSI" for more information about capacity consumption and cost implications.
- **Review capacity boundaries:** Ensure your minimum and maximum capacity settings align with your actual workload patterns.

Switching to on-demand capacity mode

For general information about switching capacity modes, see the section called "Switching capacity modes". Refer to the Service Quotas to learn about specific constraints when switching mode.

Increasing GSI throughput capacity

Use this procedure when Auto Scaling is not enabled on your GSI or you need immediate capacity increase.

1. Update the GSI's provisioned capacity using the DynamoDB console, AWS CLI, or SDK:
   - **For read capacity:** Increase the `ReadCapacityUnits` parameter for the specific GSI, which specifies the maximum number of reads the GSI can consume per second before DynamoDB throttles requests. Note that GSIs only support eventually consistent reads.
   - **For write capacity:** Increase the `WriteCapacityUnits` parameter for the specific GSI, which specifies the maximum number of writes the GSI can consume per second before DynamoDB throttles requests.
2. Ensure that the GSI's provisioned throughput capacity remains within the per-account and per-table throughput quotas.

Additional resources

- For detailed information about handling traffic spikes in DynamoDB provisioned capacity tables, including various strategies from utilizing Auto Scaling and burst capacity to strategic throttle management, see Handle traffic spikes with Amazon DynamoDB provisioned capacity.
- For information about how to use a cron expression to schedule a scaling policy, see Optimize costs by scheduling provisioned capacity for DynamoDB.
- For hands-on information about monitoring and analyzing throughput utilization patterns for your DynamoDB tables in provisioned capacity mode, see How to evaluate throughput utilization for Amazon DynamoDB tables in provisioned mode.

3- Account limits exceeded

On-demand tables do not have provisioned capacity levels to manage, but DynamoDB enforces account-level throughput limits to prevent runaway execution and ensure fair resource usage across all customers. These per-table account limits serve as adjustable safeguards, set for each account and Region combination. When your read or write consumption rate exceeds these limits, DynamoDB returns an `AccountLimitExceeded` throttling reason type in the throttling exception. The default per-table account limits automatically apply when tables do not have custom maximum throughput settings configured. You can optionally configure maximum throughput settings for finer cost control and predictability, or request quota increases through the the section called “Quotas” console if your application requirements exceed the default limits.

Account limit exceeded mitigation measures

This section provides resolution guidance for account limit throttling scenarios. Before using this guide, ensure you have identified the specific throttling reasons from your application's exception handling, and determined the Amazon Resource Name (ARN) of the affected resource. For information on retrieving throttling reasons and identifying throttled resources, see the section called “Diagnosis framework”.

Before diving into specific throttling scenarios, first determine if action is actually needed:

- **Evaluate performance impact:** Check if your application is still meeting its performance requirements despite the throttling. Many applications operate successfully at or near account limits, particularly during bulk operations or data migrations.
- **Review throttling patterns:** If throttling is intermittent and your application handles retries effectively, the current limits may be sufficient for your workload.

If your application performs acceptably even when occasionally hitting account limits, you might choose to simply monitor the situation rather than implementing immediate changes.

If you determine that throttling is causing unacceptable performance issues or reliability concerns, select a specific throttling reason below to find recommended mitigation options:

- the section called “TableReadAccountLimitExceeded”
- the section called “TableWriteAccountLimitExceeded”
- the section called “IndexReadAccountLimitExceeded”
- the section called “IndexWriteAccountLimitExceeded”

TableReadAccountLimitExceeded

When this occurs

Your table's read consumption has exceeded the account-level per-table read throughput quota for your Region. You can monitor the CloudWatch metrics in the section called “Common diagnosis and monitoring” to analyze your throttling event.

Resolution approach

Use the following steps to resolve this throttling:

- **Request quota increases:**

  Request an increase to the per-table read throughput limit (Quota code L-CF0CBE56). For detailed steps on how to submit the request, see Requesting per-table quota increases.

TableWriteAccountLimitExceeded

When this occurs

Your table's write consumption has exceeded the account-level per-table write throughput quota for your Region. You can monitor the CloudWatch metrics in the section called “Common diagnosis and monitoring” to analyze your throttling event.

Resolution approach

Use the following steps to resolve this throttling:

- **Request quota increases:** Request an increase to the per-table write throughput limit (Quota code L-AB614373). For detailed steps on how to submit the request, see Requesting per-table quota increases.

IndexReadAccountLimitExceeded

When this occurs

The read operations directed at a Global Secondary Index (GSI) consume more throughput than your account's per-table read quota allows in your current AWS Region. The account-level per-table read throughput quota applies collectively to a table and all its GSIs combined. You can monitor the CloudWatch metrics in the section called "Common diagnosis and monitoring" to analyze your throttling event.

Resolution approach

Choose the appropriate resolution based on your account's capacity distribution:

- **Request quota increases:** Request an increase to the per-table read throughput limit (Quota code L-CF0CBE56). For detailed steps on how to submit the request, see Requesting per-table quota increases.
- **Optimize GSI usage:** Review GSI design and query patterns to reduce unnecessary read capacity consumption.

IndexWriteAccountLimitExceeded

When this occurs

Write operations to your base table generate corresponding updates to a GSI that collectively exceed the account-level per-table write throughput quota for your AWS Region. Every write to a base table item that contains attributes indexed by a GSI trigger a corresponding write operation to that GSI. These combined write operations count toward your per-table write throughput quota. You can monitor the CloudWatch metrics in the section called "Common diagnosis and monitoring" to analyze the patterns and timing of these throttling events and identify which operations are causing the excessive GSI write activity.

Resolution approach

Choose the appropriate resolution based on your account's capacity distribution:

- **Request quota increases:** Request an increase to the per-table write throughput limit (Quota code L-AB614373) to accommodate higher GSI write traffic from base table operations. The per-table write throughput quota applies to the entire table, including all of its GSIs. For detailed steps on how to submit the request, see Requesting per-table quota increases.
- **Optimize GSI projections:** Review GSI projections and design to reduce write volume to GSIs.

Common diagnosis and monitoring

When troubleshooting account limit exceeded throttling events, several CloudWatch metrics can help identify whether you're hitting per-table or account-wide limits and understand your capacity distribution patterns.

Essential CloudWatch metrics

Monitor these key metrics to diagnose account limit throttling:

- **Account limit throttling events:** `ReadAccountLimitThrottleEvents` and `WriteAccountLimitThrottleEvents` track when requests are throttled due to account-level limits. `ReadThrottleEvents` and `WriteThrottleEvents` track when any read or write requests exceed the provisioned capacity.
- **Capacity consumption by resource:** `ConsumedReadCapacityUnits` and `ConsumedWriteCapacityUnits` for each table and GSI show individual resource usage.
- **Account-wide consumption:** Use CloudWatch metric math expressions to sum consumed capacity across all tables and GSIs to monitor total account usage.

Resolution procedures

Requesting per-table quota increases

If your applications need to operate beyond the current per-table throughput limits, you should submit a quota increase request using the procedure below. Each DynamoDB table in your AWS account (together with all its associated GSIs) is subject to these throughput quotas within a specific Region. These quotas represent the maximum read or write capacity that any individual table and its GSIs can collectively consume, and they apply independently to each table rather than as an aggregate across all tables in your account.

Optionally, you can also set lower limits on a per-table or per-GSI basis by configuring their maximum on-demand throughput settings.

1. Identify the specific quota that needs to be increased:
   - **Per-table read throughput limit** (Quota code L-CF0CBE56): Default 40,000 RCUs per table
   - **Per-table write throughput limit** (Quota code L-AB614373): Default 40,000 WCUs per table
2. Use the AWS Service Quotas console to request an increase:
   - Navigate to the DynamoDB service in Service Quotas
   - Find the appropriate quota using the quota code
   - Request an increase based on your projected peak usage
3. Provide justification for the increase, including:
   - Current usage patterns and peak traffic requirements
   - Business justification for the increased capacity
   - Timeline for when the increased capacity is needed

> **Note**
>
> Quota increases typically take 24-48 hours to process. Plan your requests accordingly and consider temporary mitigation strategies while waiting for approval.

Optimizing GSI projections and design

Optimize your Global Secondary Index (GSI) projections and design to reduce capacity consumption and improve performance.

Selective projection strategies

If your queries only need to access a few attributes, projecting only those attributes reduces the amount of data written to the GSI when base table items change. For details on projection types, see Projections for Global Secondary Indexes.

1. Analyze query patterns: Review your application's query patterns to identify which attributes are actually accessed through the GSI.
2. Use selective projections: Only project attributes that are actually needed in queries to reduce write volume.

3. Consider KEYS_ONLY: If your queries only need the key attributes, use KEYS_ONLY projection to minimize write volume.
4. Balance read vs. write trade-offs: Projecting fewer attributes reduces write capacity consumption but may require additional base table reads.

Sparse GSI implementation

Sparse GSIs contain only items that have the indexed attribute, rather than all items from your base table. This reduces partition density and improves performance when you frequently query specific subsets of data.

1. Design GSIs that only include items with specific attribute values.
2. Implement conditional indexing by only setting the GSI partition key attribute on items that should be indexed.
3. Use composite keys in sparse GSIs (e.g., status#timestamp) to further distribute traffic within the subset of indexed items.

For more information about implementing these strategies, see Best Practices for Using Secondary Indexes in DynamoDB.

4- On-demand maximum throughput exceeded

When you configure an on-demand table or GSI, you can optionally set a maximum throughput limit (`MaxReadRequestUnits` and `MaxWriteRequestUnits`) at the table or index level to prevent runaway costs or protect downstream systems from being overwhelmed. For more information about maximum throughput, see the section called "DynamoDB maximum throughput for on-demand tables".

When your read or write consumption exceeds these self-imposed limits, additional requests that would exceed the limit receive quick throttle responses. DynamoDB returns exceptions with a `MaxOnDemandThroughputExceeded` throttling reason type, indicating which resource has reached its throughput boundary.

On-demand maximum throughput exceeded throttling

This section provides resolution guidance for on-demand maximum throughput exceeded throttling scenarios. Before using this guide, ensure you have identified the specific throttling

reasons from your application's exception handling, and determined the Amazon Resource Name (ARN) of the affected resource. For information on retrieving throttling reasons and identifying throttled resources, see the section called “Diagnosis framework”.

Before diving into specific throttling scenarios, first consider whether action is actually needed:

- **Evaluate your maximum throughput settings:** These limits were intentionally configured to control costs or protect downstream systems. If you're receiving `MaxOnDemandThroughputExceeded` throttling events, your limits are working as designed. Consider whether increasing these limits aligns with your original cost control or system protection goals.
- **Assess application impact:** Determine if the throttling is actually causing problems for your applications or users. If your applications handle retries effectively and meets their performance requirements despite occasional throttling, maintaining your current limits might be the appropriate choice.
- **Review traffic patterns:** Analyze whether the throttling represents an expected traffic pattern or an unusual spike. For predictable, recurring traffic patterns that consistently exceed your limits, adjusting the maximum throughput settings might be warranted. For temporary spikes, implementing better request distribution strategies might be more appropriate than raising limits.

If after consideration you determine that your maximum throughput settings need adjustment, refer to the specific throttling scenarios below for targeted remediation options:

- the section called “TableReadMaxOnDemandThroughputExceeded”
- the section called “TableWriteMaxOnDemandThroughputExceeded”
- the section called “IndexReadMaxOnDemandThroughputExceeded”
- the section called “IndexWriteMaxOnDemandThroughputExceeded”

TableReadMaxOnDemandThroughputExceeded

When this occurs

Your on-demand table has exceeded its configured maximum read throughput capacity. You can monitor the CloudWatch metrics in the section called “Common diagnosis and monitoring” to analyze your throttling event.

Remediation options

Consider these steps to address your throttling events:

- **Increase maximum throughput limit:** Use the DynamoDB console, AWS CLI, or the DynamoDB `UpdateTable` API to increase the `MaxReadRequestUnits` value for the affected table, then monitor and adjust. This allows your table to handle higher read throughput before throttling occurs.
- **Remove the maximum limit:** Set `MaxReadRequestUnits` to `-1` to remove the ceiling, allowing scaling based on demand up to your account-level throughput quotas. This removes your custom limit but still maintains AWS's account-level safeguards. However, it's important to monitor spending closely after removing this limit, as your table can now consume significantly more capacity before hitting the account-level quotas.

TableWriteMaxOnDemandThroughputExceeded

When this occurs

Your on-demand table has exceeded its configured maximum write throughput capacity. You can monitor the CloudWatch metrics in the section called “Common diagnosis and monitoring” to analyze your throttling event.

Remediation options

Consider these steps to address your throttling events:

- **Increase maximum throughput limit:** Use the DynamoDB console, AWS CLI, or the DynamoDB `UpdateTable` API to increase the `MaxWriteRequestUnits` value for the affected table, then monitor and adjust.
- **Remove the maximum limit:** Set `MaxWriteRequestUnits` to `-1` to remove the ceiling, allowing scaling based on demand up to your account-level throughput quotas. This removes your custom limit but still maintains AWS's account-level safeguards. However, it's important to monitor spending closely after removing this limit, as your table can now consume significantly more capacity before hitting the account-level quotas.

IndexReadMaxOnDemandThroughputExceeded

When this occurs

Read requests to a GSI in on-demand mode have exceeded the GSI's configured maximum read throughput capacity. You can monitor the CloudWatch metrics in the section called "Common diagnosis and monitoring" to analyze your throttling event.

Remediation options

Consider these steps to address your throttling events:

- **Increase GSI maximum throughput limit:** Use the DynamoDB console, AWS CLI, or the DynamoDB `UpdateTable` API to increase the `MaxReadRequestUnits` value for the affected GSI, then monitor and adjust.
- **Remove the GSI maximum limit:** Set `MaxReadRequestUnits` to `-1` for the GSI to remove the ceiling, allowing scaling based on demand up to your account-level throughput quotas. This removes your custom limit but still maintains AWS's account-level safeguards. However, it's important to monitor spending closely after removing this limit.

IndexWriteMaxOnDemandThroughputExceeded

When this occurs

Updates to items in the base table trigger writes to a GSI in on-demand mode that exceed the GSI's configured maximum write throughput capacity, causing back-pressure throttling. You can monitor the CloudWatch metrics in the section called "Common diagnosis and monitoring" to analyze your throttling event.

Remediation options

Consider these steps to address your throttling events:

- **Increase GSI maximum throughput limit:** Use the DynamoDB console, AWS CLI, or the DynamoDB `UpdateTable` API to increase the `MaxWriteRequestUnits` value for the affected GSI, then monitor and adjust.
- **Remove the GSI maximum limit:** Set `MaxWriteRequestUnits` to `-1` for the GSI to remove the ceiling, allowing scaling based on demand up to your account-level throughput quotas. This removes your custom limit but still maintains AWS's account-level safeguards. However, it's important to monitor spending closely after removing this limit.

Common diagnosis and monitoring

When troubleshooting on-demand maximum throughput exceeded throttling events, several CloudWatch metrics can help identify the root cause and scaling patterns.

Essential CloudWatch metrics

Monitor these key metrics to diagnose on-demand maximum throughput exceeded throttling:

- **Maximum throughput throttling events:** `ReadMaxOnDemandThroughputThrottleEvents` and `WriteMaxOnDemandThroughputThrottleEvents` track when requests are throttled due to exceeding maximum limits. `ReadThrottleEvents` and `WriteThrottleEvents` track when any read or write requests exceed the provisioned capacity.
- **Current maximum throughput configured for a table or global secondary index:** `OnDemandMaxReadRequestUnits` and `OnDemandMaxWriteRequestUnits` show the current maximum capacity limits.
- **Actual capacity consumption:** `ConsumedReadCapacityUnits` and `ConsumedWriteCapacityUnits` show actual usage patterns.

Analysis approach

Follow these steps to confirm on-demand maximum throughput exceeded diagnosis:

1. Compare consumed capacity to maximum capacity limits - check if consumption consistently approaches or exceeds maximum limits.
2. Review throttling event frequency and timing to identify patterns. Look for sudden increases in consumed capacity that coincides with your throttling event.
3. Use CloudWatch Contributor Insights to identify which items or partition keys consume the most capacity.

Understanding Global Secondary Index (GSI) write throttling and back pressure in DynamoDB

GSI back-pressure throttling represents one of the most complex throttling scenarios in DynamoDB because it creates an indirect relationship between write operations and throttling—your application writes to a base table but experiences throttling due to capacity constraints on one or several indexes.

Understanding GSI back-pressure throttling

When you write to a DynamoDB table, any global secondary indexes (GSIs) on that table are updated asynchronously using an eventually consistent model. If a GSI doesn't have sufficient capacity to handle these updates, DynamoDB throttles writes to the base table to maintain data consistency. This is called *GSI back-pressure*. For more information about how GSIs work, see Global Secondary Indexes in DynamoDB.

Unlike direct table throttling where the resource being accessed is also the resource causing throttling, GSI back-pressure creates a dependency between the base table and its indexes. Even if your base table has ample capacity, writes will be throttled if any associated GSI cannot handle the update volume. This relationship is particularly important to understand because partition-level constraints apply independently to both the base table and each GSI—each has its own partition structure and corresponding throughput limits.

GSI partitioning is based on the GSI's partition key, which is often different from the base table's partition key. Even if your base table access is perfectly distributed across partitions, your GSI updates might still concentrate on specific partitions, creating hot spots in the GSI. For general best practices on partition key design for both tables and GSIs, see DynamoDB partition key design.

For example, if your base table uses `customerId` as a partition key (evenly distributed) but your GSI uses `status` as a partition key (with limited possible values like "active", "pending", "closed"), updates to items with popular status values can create GSI hot partitions even when base table access is balanced. This creates a particularly challenging scenario where your application might experience throttling due to GSI hot partitions even though both the base table and GSI have sufficient overall capacity and the base table's access pattern appears well-distributed.

Even though the throttling exception points to the GSI (via `ResourceArn`), the actual operation being throttled is the write to the base table. This can be confusing because your application is writing to the base table but receiving an exception about the GSI.

Types of GSI throttling

GSI back-pressure throttling manifests through different exception types depending on the specific capacity constraint:

- **GSI provisioned capacity exceeded:** Occurs when the GSI lacks sufficient write capacity units to handle updates from base table operations. This produces a `ProvisionedThroughputExceededException` with the reason

IndexWriteProvisionedThroughputExceeded, and the ResourceArn points directly to the specific GSI experiencing capacity constraints.

- **GSI on-demand maximum throughput exceeded:** Occurs when GSI write operations surpass configured maximum limits on on-demand tables. This produces a ThrottlingException with the reason IndexWriteMaxOnDemandThroughputExceeded, identifying the specific GSI with configured throughput restrictions.
- **GSI partition limits exceeded:** Happens when individual GSI partitions exceed their throughput limits (hot partitions), even if overall GSI capacity appears sufficient. This generates a ThrottlingException with the reason IndexWriteKeyRangeThroughputExceeded, indicating hot partition issues on the specific GSI identified in the ResourceArn. This is particularly important because GSI partition distribution may differ significantly from the base table's partition distribution, creating hot spots in the GSI even when base table access is evenly distributed.
- **GSI account limits exceeded:** Triggers when write operations to a specific GSI exceed the per-table (or any individual GSI within that table) regional throughput boundaries set at the account level. DynamoDB returns a ThrottlingException with the reason IndexWriteAccountLimitExceeded, pointing to the GSI that pushed its usage beyond account limits. This throttling occurs independently for each GSI that surpasses the limit. For information about per-table, per-account,regional, service quotas, see the section called "Quotas".

CloudWatch throttling metrics

This page provides a comprehensive guide to CloudWatch metrics specifically designed to help you identify, diagnose, and resolve throttling issues in your DynamoDB tables and indexes.

General throttling metrics

- ThrottledRequests
  - Incremented by one when any event within a request is throttled, regardless of how many individual events within that request are throttled. For example, when updating an item in a table with Global Secondary Indexes (GSIs), multiple events occur: a write operation to the base table and a write operation to each index. If any of these individual events are throttled, the ThrottledRequests metric is only incremented once.

    This behavior is important to understand when monitoring and troubleshooting DynamoDB performance, as it may mask the true extent of throttling. For more comprehensive

insights, compare the `ThrottledRequests` metric with the specific event-level metrics like `ReadThrottleEvents`, `WriteThrottleEvents`, and targeted metrics such as `ReadKeyRangeThroughputThrottleEvents` for example. The complete list of these cause-specific metrics is available in this page. Each metric corresponds to particular throttling reasons that are captured within the throttling exception. For guidance on retrieving and interpreting these reasons during throttling events, see the the section called “Diagnosing throttling” section which provides instructions for identifying and resolving the root causes of throttling issues.

- `ReadThrottleEvents`
  - Watch for requests that exceed the provisioned RCU for a table or GSI.
- `WriteThrottleEvents`
  - Watch for requests that exceed the provisioned WCU for a table or GSI.

Detailed throttling metrics by cause

On-Demand throughput throttling

- `ReadMaxOnDemandThroughputThrottleEvents`
  - Number of read requests throttled due to on-demand maximum throughput.
- `WriteMaxOnDemandThroughputThrottleEvents`
  - Number of write requests throttled due to on-demand maximum throughput.

Account-Level throttling

- `ReadAccountLimitThrottleEvents`
  - Number of read requests throttled due to account limits.
- `WriteAccountLimitThrottleEvents`
  - Number of write requests throttled due to account limits.

Partition-Level throttling

- `ReadKeyRangeThroughputThrottleEvents`
  - Number of read requests throttled due to partition limits.
- `WriteKeyRangeThroughputThrottleEvents`
  - Number of write requests throttled due to partition limits.

Capacity analysis metrics

- `OnlineIndexConsumedWriteCapacity`
  - When you add a new GSI to an existing table, DynamoDB performs a backfill operation that copies data from the base table to the new index. This process consumes write capacity units. The `OnlineIndexConsumedWriteCapacity` metric tracks this specific consumption.

    This consumption is separate from and additional to the regular write operations tracked by `ConsumedWriteCapacityUnits`. The regular `ConsumedWriteCapacityUnits` metric for a GSI does not include the write capacity consumed during the initial index creation process.
- `ProvisionedReadCapacityUnits` and `ProvisionedWriteCapacityUnits`
  - View how many provisioned read or write capacity units were consumed over the specified time period, for a table or a specified global secondary index.
  - Note that the `TableName` dimension returns `ProvisionedReadCapacityUnits` for the table only by default. To view the number of provisioned read or write capacity units for a global secondary index, you must specify both `TableName` and `GlobalSecondaryIndexName`.
- `ConsumedReadCapacityUnits` and `ConsumedWriteCapacityUnits`
  - View how many read or write capacity units were consumed over the specified time period. `ConsumedWriteCapacityUnits` does not include the write capacity consumed during the initial index creation process.

For more information on DynamoDB CloudWatch metrics, see DynamoDB Metrics and dimensions.

DynamoDB Appendix

Topics

- Troubleshooting SSL/TLS connection establishment issues with DynamoDB
- Example tables and data for use in DynamoDB
- Creating example tables and uploading data in DynamoDB
- DynamoDB example application using the AWS SDK for Python (Boto): Tic-tac-toe
- Reserved words in DynamoDB
- AWS SDK for Java 1.x examples
- AWS SDK for Go 1.x examples
- AWS SDK for Node.js 2.x examples

Troubleshooting SSL/TLS connection establishment issues with DynamoDB

Amazon DynamoDB is in the process of moving our endpoints to secure certificates signed by the Amazon Trust Services (ATS) Certificate Authority instead of third-party Certificate Authority. In December 2017, we launched the EU-WEST-3 (Paris) Region with the secure certificates issued by the Amazon Trust Services. All new regions launched after December 2017 have endpoints with the certificates issued by the Amazon Trust Services. This guide shows you how to validate and troubleshoot SSL/TLS connection issues.

Testing your application or service

Most AWS SDKs and Command Line Interfaces (CLIs) support the Amazon Trust Services Certificate Authority. If you are using a version of the AWS SDK for Python or CLI released before October 29, 2013, you must upgrade. The .NET, Java, PHP, Go, JavaScript, and C++ SDKs and CLIs do not bundle any certificates, their certificates come from the underlying operating system. The Ruby SDK has included at least one of the required CAs since June 10, 2015. Before that date, the Ruby V2 SDK did not bundle certificates. If you use an unsupported, custom, or modified version of the AWS SDK, or if you use custom trust store, you might not have the support needed for Amazon Trust Services Certificate Authority.

To validate access to DynamoDB endpoints, you will need to develop a test that accesses DynamoDB API or DynamoDB Streams API in the EU-WEST-3 region and validate that the TLS handshake succeeds. The specific endpoints you will need to access in such test are:

- DynamoDB: https://dynamodb.eu-west-3.amazonaws.com
- DynamoDB Streams: https://streams.dynamodb.eu-west-3.amazonaws.com

If your application does not support Amazon Trust Services Certificate Authority you will see one of the following failures:

- SSL/TLS Negotiation errors
- A long delay before your software receives an error indicating SSL/TLS negotiation failure. The delay time depends on the retry strategy and timeout configuration of your client.

Testing your client browser

To verify that your browser can connect to Amazon DynamoDB, open the following URL: https://dynamodb.eu-west-3.amazonaws.com. If the test is successful, you will see a message like this:

```
healthy: dynamodb.eu-west-3.amazonaws.com
```

If the test is unsuccessful, it will display an error similar to this: https://untrusted-root.badssl.com/.

Updating your software application client

Applications accessing DynamoDB or DynamoDB Streams API endpoints (whether through browsers or programmatically) will need to update the trusted CA list on the client machines if they do not already support any of the following CAs:

- Amazon Root CA 1
- Starfield Services Root Certificate Authority - G2
- Starfield Class 2 Certification Authority

If the clients already trust ANY of the above three CAs then these will trust certificates used by DynamoDB and no action is required. However, if your clients do not already trust any of the above CAs, the HTTPS connections to the DynamoDB or DynamoDB Streams APIs will fail. For more

information, please visit this blog post: https://aws.amazon.com/blogs/security/how-to-prepare-for-aws-move-to-its-own-certificate-authority/.

Updating your client browser

You can update the certificate bundle in your browser simply by updating your browser. Instructions for the most common browsers can be found on the browsers' websites:

- Chrome: https://support.google.com/chrome/answer/95414?hl=en
- Firefox: https://support.mozilla.org/en-US/kb/update-firefox-latest-version
- Safari: https://support.apple.com/en-us/HT204416
- Internet Explorer: https://support.microsoft.com/en-us/help/17295/windows-internet-explorer-which-version#ie=other

Manually updating your certificate bundle

If you can't access the DynamoDB API or DynamoDB Streams API then you need to update your certificate bundle. To do this, you need to import at least one of the required CAs. You can find these at https://www.amazontrust.com/repository/.

The following operating systems and programming languages support Amazon Trust Services certificates:

- Microsoft Windows versions that have January 2005 or later updates installed, Windows Vista, Windows 7, Windows Server 2008, and newer versions.
- MacOS X 10.4 with Java for MacOS X 10.4 Release 5, MacOS X 10.5 and newer versions.
- Red Hat Enterprise Linux 5 (March 2007), Linux 6, and Linux 7 and CentOS 5, CentOS 6, and CentOS 7
- Ubuntu 8.10
- Debian 5.0
- Amazon Linux (all versions)
- Java 1.4.2_12, Java 5 update 2, and all newer versions, including Java 6, Java 7, and Java 8

If you're still unable to connect, consult your software documentation, OS Vendor, or contact with AWS Support https://aws.amazon.com/support for further assistance.

Example tables and data for use in DynamoDB

The *Amazon DynamoDB Developer Guide* uses sample tables to illustrate various aspects of DynamoDB.

Table name	Primary key
ProductCatalog	Simple primary key: • `Id` (Number)
Forum	Simple primary key: • `Name` (String)
Thread	Composite primary key: • `ForumName` (String) • `Subject` (String)
Reply	Composite primary key: • `Id` (String) • `ReplyDateTime` (String)

The *Reply* table has a global secondary index named *PostedBy-Message-Index*. This index will facilitate queries on two non-key attributes of the *Reply* table.

Index name	Primary key
PostedBy-Message-Index	Composite primary key: • `PostedBy` (String) • `Message` (String)

For more information about these tables, see Step 1: Create a table in DynamoDB and Step 2: Write data to a DynamoDB table.

Sample data files

Topics

- ProductCatalog sample data
- Forum sample data
- Thread sample data
- Reply sample data

The following sections show the sample data files that are used for loading the *ProductCatalog*, *Forum*, *Thread* and *Reply* tables.

Each data file contains multiple `PutRequest` elements, each of which contain a single item. These `PutRequest` elements are used as input to the `BatchWriteItem` operation, using the AWS Command Line Interface (AWS CLI).

ProductCatalog sample data

```
{
    "ProductCatalog": [
        {
            "PutRequest": {
                "Item": {
                    "Id": {
                        "N": "101"
                    },
                    "Title": {
                        "S": "Book 101 Title"
                    },
                    "ISBN": {
                        "S": "111-1111111111"
                    },
                    "Authors": {
                        "L": [
                            {
                                "S": "Author1"
                            }
                        ]
                    },
                    "Price": {
                        "N": "2"
```

```
                    },
                    "Dimensions": {
                        "S": "8.5 x 11.0 x 0.5"
                    },
                    "PageCount": {
                        "N": "500"
                    },
                    "InPublication": {
                        "BOOL": true
                    },
                    "ProductCategory": {
                        "S": "Book"
                    }
                }
            }
        },
        {
            "PutRequest": {
                "Item": {
                    "Id": {
                        "N": "102"
                    },
                    "Title": {
                        "S": "Book 102 Title"
                    },
                    "ISBN": {
                        "S": "222-2222222222"
                    },
                    "Authors": {
                        "L": [
                            {
                                "S": "Author1"
                            },
                            {
                                "S": "Author2"
                            }
                        ]
                    },
                    "Price": {
                        "N": "20"
                    },
                    "Dimensions": {
                        "S": "8.5 x 11.0 x 0.8"
                    },
```

```
                    "PageCount": {
                        "N": "600"
                    },
                    "InPublication": {
                        "BOOL": true
                    },
                    "ProductCategory": {
                        "S": "Book"
                    }
                }
            }
        },
        {
            "PutRequest": {
                "Item": {
                    "Id": {
                        "N": "103"
                    },
                    "Title": {
                        "S": "Book 103 Title"
                    },
                    "ISBN": {
                        "S": "333-3333333333"
                    },
                    "Authors": {
                        "L": [
                            {
                                "S": "Author1"
                            },
                            {
                                "S": "Author2"
                            }
                        ]
                    },
                    "Price": {
                        "N": "2000"
                    },
                    "Dimensions": {
                        "S": "8.5 x 11.0 x 1.5"
                    },
                    "PageCount": {
                        "N": "600"
                    },
                    "InPublication": {
```

```
                    "BOOL": false
                },
                "ProductCategory": {
                    "S": "Book"
                }
            }
        },
        {
            "PutRequest": {
                "Item": {
                    "Id": {
                        "N": "201"
                    },
                    "Title": {
                        "S": "18-Bike-201"
                    },
                    "Description": {
                        "S": "201 Description"
                    },
                    "BicycleType": {
                        "S": "Road"
                    },
                    "Brand": {
                        "S": "Mountain A"
                    },
                    "Price": {
                        "N": "100"
                    },
                    "Color": {
                        "L": [
                            {
                                "S": "Red"
                            },
                            {
                                "S": "Black"
                            }
                        ]
                    },
                    "ProductCategory": {
                        "S": "Bicycle"
                    }
                }
            }
```

```
        },
        {
            "PutRequest": {
                "Item": {
                    "Id": {
                        "N": "202"
                    },
                    "Title": {
                        "S": "21-Bike-202"
                    },
                    "Description": {
                        "S": "202 Description"
                    },
                    "BicycleType": {
                        "S": "Road"
                    },
                    "Brand": {
                        "S": "Brand-Company A"
                    },
                    "Price": {
                        "N": "200"
                    },
                    "Color": {
                        "L": [
                            {
                                "S": "Green"
                            },
                            {
                                "S": "Black"
                            }
                        ]
                    },
                    "ProductCategory": {
                        "S": "Bicycle"
                    }
                }
            }
        },
        {
            "PutRequest": {
                "Item": {
                    "Id": {
                        "N": "203"
                    },
```

```
                    "Title": {
                        "S": "19-Bike-203"
                    },
                    "Description": {
                        "S": "203 Description"
                    },
                    "BicycleType": {
                        "S": "Road"
                    },
                    "Brand": {
                        "S": "Brand-Company B"
                    },
                    "Price": {
                        "N": "300"
                    },
                    "Color": {
                        "L": [
                            {
                                "S": "Red"
                            },
                            {
                                "S": "Green"
                            },
                            {
                                "S": "Black"
                            }
                        ]
                    },
                    "ProductCategory": {
                        "S": "Bicycle"
                    }
                }
            }
        },
        {
            "PutRequest": {
                "Item": {
                    "Id": {
                        "N": "204"
                    },
                    "Title": {
                        "S": "18-Bike-204"
                    },
                    "Description": {
```

```
                    "S": "204 Description"
                },
                "BicycleType": {
                    "S": "Mountain"
                },
                "Brand": {
                    "S": "Brand-Company B"
                },
                "Price": {
                    "N": "400"
                },
                "Color": {
                    "L": [
                        {
                            "S": "Red"
                        }
                    ]
                },
                "ProductCategory": {
                    "S": "Bicycle"
                }
            }
        }
    },
    {
        "PutRequest": {
            "Item": {
                "Id": {
                    "N": "205"
                },
                "Title": {
                    "S": "18-Bike-204"
                },
                "Description": {
                    "S": "205 Description"
                },
                "BicycleType": {
                    "S": "Hybrid"
                },
                "Brand": {
                    "S": "Brand-Company C"
                },
                "Price": {
                    "N": "500"
```

```
                    },
                    "Color": {
                        "L": [
                            {
                                "S": "Red"
                            },
                            {
                                "S": "Black"
                            }
                        ]
                    },
                    "ProductCategory": {
                        "S": "Bicycle"
                    }
                }
            }
        }
    ]
}
```

Forum sample data

```
{
    "Forum": [
        {
            "PutRequest": {
                "Item": {
                    "Name": {"S":"Amazon DynamoDB"},
                    "Category": {"S":"Amazon Web Services"},
                    "Threads": {"N":"2"},
                    "Messages": {"N":"4"},
                    "Views": {"N":"1000"}
                }
            }
        },
        {
            "PutRequest": {
                "Item": {
                    "Name": {"S":"Amazon S3"},
                    "Category": {"S":"Amazon Web Services"}
                }
            }
        }
```

```
    ]
}
```

Thread sample data

```
{
    "Thread": [
        {
            "PutRequest": {
                "Item": {
                    "ForumName": {
                        "S": "Amazon DynamoDB"
                    },
                    "Subject": {
                        "S": "DynamoDB Thread 1"
                    },
                    "Message": {
                        "S": "DynamoDB thread 1 message"
                    },
                    "LastPostedBy": {
                        "S": "User A"
                    },
                    "LastPostedDateTime": {
                        "S": "2015-09-22T19:58:22.514Z"
                    },
                    "Views": {
                        "N": "0"
                    },
                    "Replies": {
                        "N": "0"
                    },
                    "Answered": {
                        "N": "0"
                    },
                    "Tags": {
                        "L": [
                            {
                                "S": "index"
                            },
                            {
                                "S": "primarykey"
                            },
                            {
```

```
                                "S": "table"
                            }
                        ]
                    }
                }
            }
        },
        {
            "PutRequest": {
                "Item": {
                    "ForumName": {
                        "S": "Amazon DynamoDB"
                    },
                    "Subject": {
                        "S": "DynamoDB Thread 2"
                    },
                    "Message": {
                        "S": "DynamoDB thread 2 message"
                    },
                    "LastPostedBy": {
                        "S": "User A"
                    },
                    "LastPostedDateTime": {
                        "S": "2015-09-15T19:58:22.514Z"
                    },
                    "Views": {
                        "N": "3"
                    },
                    "Replies": {
                        "N": "0"
                    },
                    "Answered": {
                        "N": "0"
                    },
                    "Tags": {
                        "L": [
                            {
                                "S": "items"
                            },
                            {
                                "S": "attributes"
                            },
                            {
                                "S": "throughput"
```

```
                        }
                    ]
                }
            }
        },
        {
            "PutRequest": {
                "Item": {
                    "ForumName": {
                        "S": "Amazon S3"
                    },
                    "Subject": {
                        "S": "S3 Thread 1"
                    },
                    "Message": {
                        "S": "S3 thread 1 message"
                    },
                    "LastPostedBy": {
                        "S": "User A"
                    },
                    "LastPostedDateTime": {
                        "S": "2015-09-29T19:58:22.514Z"
                    },
                    "Views": {
                        "N": "0"
                    },
                    "Replies": {
                        "N": "0"
                    },
                    "Answered": {
                        "N": "0"
                    },
                    "Tags": {
                        "L": [
                            {
                                "S": "largeobjects"
                            },
                            {
                                "S": "multipart upload"
                            }
                        ]
                    }
                }
```

```
            }
        }
    ]
}
```

Reply sample data

```
{
    "Reply": [
        {
            "PutRequest": {
                "Item": {
                    "Id": {
                        "S": "Amazon DynamoDB#DynamoDB Thread 1"
                    },
                    "ReplyDateTime": {
                        "S": "2015-09-15T19:58:22.947Z"
                    },
                    "Message": {
                        "S": "DynamoDB Thread 1 Reply 1 text"
                    },
                    "PostedBy": {
                        "S": "User A"
                    }
                }
            }
        },
        {
            "PutRequest": {
                "Item": {
                    "Id": {
                        "S": "Amazon DynamoDB#DynamoDB Thread 1"
                    },
                    "ReplyDateTime": {
                        "S": "2015-09-22T19:58:22.947Z"
                    },
                    "Message": {
                        "S": "DynamoDB Thread 1 Reply 2 text"
                    },
                    "PostedBy": {
                        "S": "User B"
                    }
                }
```

```
            }
        },
        {
            "PutRequest": {
                "Item": {
                    "Id": {
                        "S": "Amazon DynamoDB#DynamoDB Thread 2"
                    },
                    "ReplyDateTime": {
                        "S": "2015-09-29T19:58:22.947Z"
                    },
                    "Message": {
                        "S": "DynamoDB Thread 2 Reply 1 text"
                    },
                    "PostedBy": {
                        "S": "User A"
                    }
                }
            }
        },
        {
            "PutRequest": {
                "Item": {
                    "Id": {
                        "S": "Amazon DynamoDB#DynamoDB Thread 2"
                    },
                    "ReplyDateTime": {
                        "S": "2015-10-05T19:58:22.947Z"
                    },
                    "Message": {
                        "S": "DynamoDB Thread 2 Reply 2 text"
                    },
                    "PostedBy": {
                        "S": "User A"
                    }
                }
            }
        }
    ]
}
```

Creating example tables and uploading data in DynamoDB

This appendix provides code to both create the tables and add data programmatically.

Topics

- Creating example tables and uploading data using the AWS SDK for Java
- Creating example tables and uploading data using the AWS SDK for .NET

Creating example tables and uploading data using the AWS SDK for Java

The following Java code example creates tables and uploads data to the tables. For step-by-step instructions to run this code using Eclipse, see Java code examples.

```
package com.amazonaws.codesamples;

import java.text.SimpleDateFormat;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Date;
import java.util.HashSet;
import java.util.TimeZone;

import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;
import com.amazonaws.services.dynamodbv2.document.DynamoDB;
import com.amazonaws.services.dynamodbv2.document.Item;
import com.amazonaws.services.dynamodbv2.document.Table;
import com.amazonaws.services.dynamodbv2.model.AttributeDefinition;
import com.amazonaws.services.dynamodbv2.model.CreateTableRequest;
import com.amazonaws.services.dynamodbv2.model.KeySchemaElement;
import com.amazonaws.services.dynamodbv2.model.KeyType;
import com.amazonaws.services.dynamodbv2.model.LocalSecondaryIndex;
import com.amazonaws.services.dynamodbv2.model.Projection;
import com.amazonaws.services.dynamodbv2.model.ProjectionType;
import com.amazonaws.services.dynamodbv2.model.ProvisionedThroughput;

public class CreateTablesLoadData {
```

```
    static AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();
    static DynamoDB dynamoDB = new DynamoDB(client);

    static SimpleDateFormat dateFormatter = new SimpleDateFormat("yyyy-MM-
dd'T'HH:mm:ss.SSS'Z'");

    static String productCatalogTableName = "ProductCatalog";
    static String forumTableName = "Forum";
    static String threadTableName = "Thread";
    static String replyTableName = "Reply";

    public static void main(String[] args) throws Exception {

        try {

            deleteTable(productCatalogTableName);
            deleteTable(forumTableName);
            deleteTable(threadTableName);
            deleteTable(replyTableName);

            // Parameter1: table name
            // Parameter2: reads per second
            // Parameter3: writes per second
            // Parameter4/5: partition key and data type
            // Parameter6/7: sort key and data type (if applicable)

            createTable(productCatalogTableName, 10L, 5L, "Id", "N");
            createTable(forumTableName, 10L, 5L, "Name", "S");
            createTable(threadTableName, 10L, 5L, "ForumName", "S", "Subject", "S");
            createTable(replyTableName, 10L, 5L, "Id", "S", "ReplyDateTime", "S");

            loadSampleProducts(productCatalogTableName);
            loadSampleForums(forumTableName);
            loadSampleThreads(threadTableName);
            loadSampleReplies(replyTableName);

        } catch (Exception e) {
            System.err.println("Program failed:");
            System.err.println(e.getMessage());
        }
        System.out.println("Success.");
    }

    private static void deleteTable(String tableName) {
```

```
        Table table = dynamoDB.getTable(tableName);
        try {
            System.out.println("Issuing DeleteTable request for " + tableName);
            table.delete();
            System.out.println("Waiting for " + tableName + " to be deleted...this may
take a while...");
            table.waitForDelete();

        } catch (Exception e) {
            System.err.println("DeleteTable request failed for " + tableName);
            System.err.println(e.getMessage());
        }
    }

    private static void createTable(String tableName, long readCapacityUnits, long
writeCapacityUnits,
            String partitionKeyName, String partitionKeyType) {

        createTable(tableName, readCapacityUnits, writeCapacityUnits, partitionKeyName,
partitionKeyType, null, null);
    }

    private static void createTable(String tableName, long readCapacityUnits, long
writeCapacityUnits,
            String partitionKeyName, String partitionKeyType, String sortKeyName,
String sortKeyType) {

        try {

            ArrayList<KeySchemaElement> keySchema = new ArrayList<KeySchemaElement>();
            keySchema.add(new
KeySchemaElement().withAttributeName(partitionKeyName).withKeyType(KeyType.HASH)); //
Partition

                            // key

            ArrayList<AttributeDefinition> attributeDefinitions = new
ArrayList<AttributeDefinition>();
            attributeDefinitions
                    .add(new AttributeDefinition().withAttributeName(partitionKeyName)
                            .withAttributeType(partitionKeyType));

            if (sortKeyName != null) {
```

```
                keySchema.add(new
KeySchemaElement().withAttributeName(sortKeyName).withKeyType(KeyType.RANGE)); // Sort

                          // key
                attributeDefinitions
                        .add(new
AttributeDefinition().withAttributeName(sortKeyName).withAttributeType(sortKeyType));
            }

            CreateTableRequest request = new
CreateTableRequest().withTableName(tableName).withKeySchema(keySchema)
                    .withProvisionedThroughput(new
ProvisionedThroughput().withReadCapacityUnits(readCapacityUnits)
                            .withWriteCapacityUnits(writeCapacityUnits));

            // If this is the Reply table, define a local secondary index
            if (replyTableName.equals(tableName)) {

                attributeDefinitions
                        .add(new
AttributeDefinition().withAttributeName("PostedBy").withAttributeType("S"));

                ArrayList<LocalSecondaryIndex> localSecondaryIndexes = new
ArrayList<LocalSecondaryIndex>();
                localSecondaryIndexes.add(new
LocalSecondaryIndex().withIndexName("PostedBy-Index")
                        .withKeySchema(
                                new
KeySchemaElement().withAttributeName(partitionKeyName).withKeyType(KeyType.HASH), //
Partition

                                // key
                                new
KeySchemaElement().withAttributeName("PostedBy").withKeyType(KeyType.RANGE)) // Sort

                          // key
                        .withProjection(new
Projection().withProjectionType(ProjectionType.KEYS_ONLY)));

                request.setLocalSecondaryIndexes(localSecondaryIndexes);
            }

            request.setAttributeDefinitions(attributeDefinitions);
```

```
            System.out.println("Issuing CreateTable request for " + tableName);
            Table table = dynamoDB.createTable(request);
            System.out.println("Waiting for " + tableName + " to be created...this may
take a while...");
            table.waitForActive();

        } catch (Exception e) {
            System.err.println("CreateTable request failed for " + tableName);
            System.err.println(e.getMessage());
        }
    }

    private static void loadSampleProducts(String tableName) {

        Table table = dynamoDB.getTable(tableName);

        try {

            System.out.println("Adding data to " + tableName);

            Item item = new Item().withPrimaryKey("Id", 101).withString("Title", "Book
101 Title")
                    .withString("ISBN", "111-1111111111")
                    .withStringSet("Authors", new
HashSet<String>(Arrays.asList("Author1"))).withNumber("Price", 2)
                    .withString("Dimensions", "8.5 x 11.0 x
0.5").withNumber("PageCount", 500)
                    .withBoolean("InPublication", true).withString("ProductCategory",
"Book");
            table.putItem(item);

            item = new Item().withPrimaryKey("Id", 102).withString("Title", "Book 102
Title")
                    .withString("ISBN", "222-2222222222")
                    .withStringSet("Authors", new
HashSet<String>(Arrays.asList("Author1", "Author2")))
                    .withNumber("Price", 20).withString("Dimensions", "8.5 x 11.0 x
0.8").withNumber("PageCount", 600)
                    .withBoolean("InPublication", true).withString("ProductCategory",
"Book");
            table.putItem(item);

            item = new Item().withPrimaryKey("Id", 103).withString("Title", "Book 103
Title")
```

```
                    .withString("ISBN", "333-3333333333")
                    .withStringSet("Authors", new
 HashSet<String>(Arrays.asList("Author1", "Author2")))
                    // Intentional. Later we'll run Scan to find price error. Find
                    // items > 1000 in price.
                    .withNumber("Price", 2000).withString("Dimensions", "8.5 x 11.0 x
 1.5").withNumber("PageCount", 600)
                    .withBoolean("InPublication", false).withString("ProductCategory",
 "Book");
            table.putItem(item);

            // Add bikes.

            item = new Item().withPrimaryKey("Id", 201).withString("Title", "18-
Bike-201")
                    // Size, followed by some title.
                    .withString("Description", "201
 Description").withString("BicycleType", "Road")
                    .withString("Brand", "Mountain A")
                    // Trek, Specialized.
                    .withNumber("Price", 100).withStringSet("Color", new
 HashSet<String>(Arrays.asList("Red", "Black")))
                    .withString("ProductCategory", "Bicycle");
            table.putItem(item);

            item = new Item().withPrimaryKey("Id", 202).withString("Title", "21-
Bike-202")
                    .withString("Description", "202
 Description").withString("BicycleType", "Road")
                    .withString("Brand", "Brand-Company A").withNumber("Price", 200)
                    .withStringSet("Color", new HashSet<String>(Arrays.asList("Green",
 "Black")))
                    .withString("ProductCategory", "Bicycle");
            table.putItem(item);

            item = new Item().withPrimaryKey("Id", 203).withString("Title", "19-
Bike-203")
                    .withString("Description", "203
 Description").withString("BicycleType", "Road")
                    .withString("Brand", "Brand-Company B").withNumber("Price", 300)
                    .withStringSet("Color", new HashSet<String>(Arrays.asList("Red",
 "Green", "Black")))
                    .withString("ProductCategory", "Bicycle");
            table.putItem(item);
```

```
            item = new Item().withPrimaryKey("Id", 204).withString("Title", "18-
Bike-204")
                    .withString("Description", "204
 Description").withString("BicycleType", "Mountain")
                    .withString("Brand", "Brand-Company B").withNumber("Price", 400)
                    .withStringSet("Color", new HashSet<String>(Arrays.asList("Red")))
                    .withString("ProductCategory", "Bicycle");
            table.putItem(item);

            item = new Item().withPrimaryKey("Id", 205).withString("Title", "20-
Bike-205")
                    .withString("Description", "205
 Description").withString("BicycleType", "Hybrid")
                    .withString("Brand", "Brand-Company C").withNumber("Price", 500)
                    .withStringSet("Color", new HashSet<String>(Arrays.asList("Red",
 "Black")))
                    .withString("ProductCategory", "Bicycle");
            table.putItem(item);

        } catch (Exception e) {
            System.err.println("Failed to create item in " + tableName);
            System.err.println(e.getMessage());
        }

    }

    private static void loadSampleForums(String tableName) {

        Table table = dynamoDB.getTable(tableName);

        try {

            System.out.println("Adding data to " + tableName);

            Item item = new Item().withPrimaryKey("Name", "Amazon DynamoDB")
                    .withString("Category", "Amazon Web
 Services").withNumber("Threads", 2).withNumber("Messages", 4)
                    .withNumber("Views", 1000);
            table.putItem(item);

            item = new Item().withPrimaryKey("Name", "Amazon
 S3").withString("Category", "Amazon Web Services")
                    .withNumber("Threads", 0);
```

```
            table.putItem(item);

        } catch (Exception e) {
            System.err.println("Failed to create item in " + tableName);
            System.err.println(e.getMessage());
        }
    }

    private static void loadSampleThreads(String tableName) {
        try {
            long time1 = (new Date()).getTime() - (7 * 24 * 60 * 60 * 1000); // 7
            // days
            // ago
            long time2 = (new Date()).getTime() - (14 * 24 * 60 * 60 * 1000); // 14
            // days
            // ago
            long time3 = (new Date()).getTime() - (21 * 24 * 60 * 60 * 1000); // 21
            // days
            // ago

            Date date1 = new Date();
            date1.setTime(time1);

            Date date2 = new Date();
            date2.setTime(time2);

            Date date3 = new Date();
            date3.setTime(time3);

            dateFormatter.setTimeZone(TimeZone.getTimeZone("UTC"));

            Table table = dynamoDB.getTable(tableName);

            System.out.println("Adding data to " + tableName);

            Item item = new Item().withPrimaryKey("ForumName", "Amazon DynamoDB")
                    .withString("Subject", "DynamoDB Thread 1").withString("Message",
"DynamoDB thread 1 message")
                    .withString("LastPostedBy", "User
A").withString("LastPostedDateTime", dateFormatter.format(date2))
                    .withNumber("Views", 0).withNumber("Replies",
0).withNumber("Answered", 0)
                    .withStringSet("Tags", new HashSet<String>(Arrays.asList("index",
"primarykey", "table")));
```

```
            table.putItem(item);

            item = new Item().withPrimaryKey("ForumName", "Amazon
DynamoDB").withString("Subject", "DynamoDB Thread 2")
                    .withString("Message", "DynamoDB thread 2
message").withString("LastPostedBy", "User A")
                    .withString("LastPostedDateTime",
dateFormatter.format(date3)).withNumber("Views", 0)
                    .withNumber("Replies", 0).withNumber("Answered", 0)
                    .withStringSet("Tags", new HashSet<String>(Arrays.asList("index",
"partitionkey", "sortkey")));
            table.putItem(item);

            item = new Item().withPrimaryKey("ForumName", "Amazon
S3").withString("Subject", "S3 Thread 1")
                    .withString("Message", "S3 Thread 3
message").withString("LastPostedBy", "User A")
                    .withString("LastPostedDateTime",
dateFormatter.format(date1)).withNumber("Views", 0)
                    .withNumber("Replies", 0).withNumber("Answered", 0)
                    .withStringSet("Tags", new
HashSet<String>(Arrays.asList("largeobjects", "multipart upload")));
            table.putItem(item);

        } catch (Exception e) {
            System.err.println("Failed to create item in " + tableName);
            System.err.println(e.getMessage());
        }

    }

    private static void loadSampleReplies(String tableName) {
        try {
            // 1 day ago
            long time0 = (new Date()).getTime() - (1 * 24 * 60 * 60 * 1000);
            // 7 days ago
            long time1 = (new Date()).getTime() - (7 * 24 * 60 * 60 * 1000);
            // 14 days ago
            long time2 = (new Date()).getTime() - (14 * 24 * 60 * 60 * 1000);
            // 21 days ago
            long time3 = (new Date()).getTime() - (21 * 24 * 60 * 60 * 1000);

            Date date0 = new Date();
            date0.setTime(time0);
```

```
            Date date1 = new Date();
            date1.setTime(time1);

            Date date2 = new Date();
            date2.setTime(time2);

            Date date3 = new Date();
            date3.setTime(time3);

            dateFormatter.setTimeZone(TimeZone.getTimeZone("UTC"));

            Table table = dynamoDB.getTable(tableName);

            System.out.println("Adding data to " + tableName);

            // Add threads.

            Item item = new Item().withPrimaryKey("Id", "Amazon DynamoDB#DynamoDB
Thread 1")
                    .withString("ReplyDateTime", (dateFormatter.format(date3)))
                    .withString("Message", "DynamoDB Thread 1 Reply 1
text").withString("PostedBy", "User A");
            table.putItem(item);

            item = new Item().withPrimaryKey("Id", "Amazon DynamoDB#DynamoDB Thread 1")
                    .withString("ReplyDateTime", dateFormatter.format(date2))
                    .withString("Message", "DynamoDB Thread 1 Reply 2
text").withString("PostedBy", "User B");
            table.putItem(item);

            item = new Item().withPrimaryKey("Id", "Amazon DynamoDB#DynamoDB Thread 2")
                    .withString("ReplyDateTime", dateFormatter.format(date1))
                    .withString("Message", "DynamoDB Thread 2 Reply 1
text").withString("PostedBy", "User A");
            table.putItem(item);

            item = new Item().withPrimaryKey("Id", "Amazon DynamoDB#DynamoDB Thread 2")
                    .withString("ReplyDateTime", dateFormatter.format(date0))
                    .withString("Message", "DynamoDB Thread 2 Reply 2
text").withString("PostedBy", "User A");
            table.putItem(item);

        } catch (Exception e) {
```

```
            System.err.println("Failed to create item in " + tableName);
            System.err.println(e.getMessage());

        }
    }

}
```

Creating example tables and uploading data using the AWS SDK for .NET

The following C# code example creates tables and uploads data to the tables. For step-by-step instructions to run this code in Visual Studio, see .NET code examples.

```
using System;
using System.Collections.Generic;
using Amazon.DynamoDBv2;
using Amazon.DynamoDBv2.DocumentModel;
using Amazon.DynamoDBv2.Model;
using Amazon.Runtime;
using Amazon.SecurityToken;

namespace com.amazonaws.codesamples
{
    class CreateTablesLoadData
    {
        private static AmazonDynamoDBClient client = new AmazonDynamoDBClient();

        static void Main(string[] args)
        {
            try
            {
                //DeleteAllTables(client);
                DeleteTable("ProductCatalog");
                DeleteTable("Forum");
                DeleteTable("Thread");
                DeleteTable("Reply");

                // Create tables (using the AWS SDK for .NET low-level API).
                CreateTableProductCatalog();
                CreateTableForum();
```

```
            CreateTableThread(); // ForumTitle, Subject */
            CreateTableReply();

            // Load data (using the .NET SDK document API)
            LoadSampleProducts();
            LoadSampleForums();
            LoadSampleThreads();
            LoadSampleReplies();
            Console.WriteLine("Sample complete!");
            Console.WriteLine("Press ENTER to continue");
            Console.ReadLine();
        }
        catch (AmazonServiceException e) { Console.WriteLine(e.Message); }
        catch (Exception e) { Console.WriteLine(e.Message); }
    }

    private static void DeleteTable(string tableName)
    {
        try
        {
            var deleteTableResponse = client.DeleteTable(new DeleteTableRequest()
            {
                TableName = tableName
            });
            WaitTillTableDeleted(client, tableName, deleteTableResponse);
        }
        catch (ResourceNotFoundException)
        {
            // There is no such table.
        }
    }

    private static void CreateTableProductCatalog()
    {
        string tableName = "ProductCatalog";

        var response = client.CreateTable(new CreateTableRequest
        {
            TableName = tableName,
            AttributeDefinitions = new List<AttributeDefinition>()
                        {
                            new AttributeDefinition
                            {
                                AttributeName = "Id",
```

```
                                AttributeType = "N"
                            }
                        },
            KeySchema = new List<KeySchemaElement>()
                        {
                            new KeySchemaElement
                            {
                                AttributeName = "Id",
                                KeyType = "HASH"
                            }
                        },
            ProvisionedThroughput = new ProvisionedThroughput
            {
                ReadCapacityUnits = 10,
                WriteCapacityUnits = 5
            }
        });

        WaitTillTableCreated(client, tableName, response);
    }

    private static void CreateTableForum()
    {
        string tableName = "Forum";

        var response = client.CreateTable(new CreateTableRequest
        {
            TableName = tableName,
            AttributeDefinitions = new List<AttributeDefinition>()
                        {
                            new AttributeDefinition
                            {
                                AttributeName = "Name",
                                AttributeType = "S"
                            }
                        },
            KeySchema = new List<KeySchemaElement>()
                        {
                            new KeySchemaElement
                            {
                                AttributeName = "Name", // forum Title
                                KeyType = "HASH"
                            }
                        },
```

```
            ProvisionedThroughput = new ProvisionedThroughput
            {
                ReadCapacityUnits = 10,
                WriteCapacityUnits = 5
            }
        });

        WaitTillTableCreated(client, tableName, response);
    }

    private static void CreateTableThread()
    {
        string tableName = "Thread";

        var response = client.CreateTable(new CreateTableRequest
        {
            TableName = tableName,
            AttributeDefinitions = new List<AttributeDefinition>()
                        {
                            new AttributeDefinition
                            {
                                AttributeName = "ForumName", // Hash attribute
                                AttributeType = "S"
                            },
                            new AttributeDefinition
                            {
                                AttributeName = "Subject",
                                AttributeType = "S"
                            }
                        },
            KeySchema = new List<KeySchemaElement>()
                        {
                            new KeySchemaElement
                            {
                                AttributeName = "ForumName", // Hash attribute
                                KeyType = "HASH"
                            },
                            new KeySchemaElement
                            {
                                AttributeName = "Subject", // Range attribute
                                KeyType = "RANGE"
                            }
                        },
            ProvisionedThroughput = new ProvisionedThroughput
```

```
            {
                ReadCapacityUnits = 10,
                WriteCapacityUnits = 5
            }
        });

        WaitTillTableCreated(client, tableName, response);
    }

    private static void CreateTableReply()
    {
        string tableName = "Reply";
        var response = client.CreateTable(new CreateTableRequest
        {
            TableName = tableName,
            AttributeDefinitions = new List<AttributeDefinition>()
                        {
                            new AttributeDefinition
                            {
                                AttributeName = "Id",
                                AttributeType = "S"
                            },
                            new AttributeDefinition
                            {
                                AttributeName = "ReplyDateTime",
                                AttributeType = "S"
                            },
                            new AttributeDefinition
                            {
                                AttributeName = "PostedBy",
                                AttributeType = "S"
                            }
                        },
            KeySchema = new List<KeySchemaElement>()
                        {
                            new KeySchemaElement()
                            {
                                AttributeName = "Id",
                                KeyType = "HASH"
                            },
                            new KeySchemaElement()
                            {
                                AttributeName = "ReplyDateTime",
                                KeyType = "RANGE"
```

```
                                }
                            },
                LocalSecondaryIndexes = new List<LocalSecondaryIndex>()
                            {
                                new LocalSecondaryIndex()
                                {
                                    IndexName = "PostedBy_index",


                                    KeySchema = new List<KeySchemaElement>() {
                                        new KeySchemaElement() {
                                            AttributeName = "Id", KeyType = "HASH"
                                        },
                                        new KeySchemaElement() {
                                            AttributeName = "PostedBy", KeyType =
"RANGE"
                                        }
                                    },
                                    Projection = new Projection() {
                                        ProjectionType = ProjectionType.KEYS_ONLY
                                    }
                                }
                            },
                ProvisionedThroughput = new ProvisionedThroughput
                {
                    ReadCapacityUnits = 10,
                    WriteCapacityUnits = 5
                }
            });

            WaitTillTableCreated(client, tableName, response);
        }

        private static void WaitTillTableCreated(AmazonDynamoDBClient client, string
tableName,
                                 CreateTableResponse response)
        {
            var tableDescription = response.TableDescription;

            string status = tableDescription.TableStatus;

            Console.WriteLine(tableName + " - " + status);

            // Let us wait until table is created. Call DescribeTable.
```

```
            while (status != "ACTIVE")
            {
                System.Threading.Thread.Sleep(5000); // Wait 5 seconds.
                try
                {
                    var res = client.DescribeTable(new DescribeTableRequest
                    {
                        TableName = tableName
                    });
                    Console.WriteLine("Table name: {0}, status: {1}",
res.Table.TableName,
                              res.Table.TableStatus);
                    status = res.Table.TableStatus;
                }
                // Try-catch to handle potential eventual-consistency issue.
                catch (ResourceNotFoundException)
                { }
            }
        }

        private static void WaitTillTableDeleted(AmazonDynamoDBClient client, string
tableName,
                             DeleteTableResponse response)
        {
            var tableDescription = response.TableDescription;

            string status = tableDescription.TableStatus;

            Console.WriteLine(tableName + " - " + status);

            // Let us wait until table is created. Call DescribeTable
            try
            {
                while (status == "DELETING")
                {
                    System.Threading.Thread.Sleep(5000); // wait 5 seconds

                    var res = client.DescribeTable(new DescribeTableRequest
                    {
                        TableName = tableName
                    });
                    Console.WriteLine("Table name: {0}, status: {1}",
res.Table.TableName,
                              res.Table.TableStatus);
```

```
                    status = res.Table.TableStatus;
                }
            }
            catch (ResourceNotFoundException)
            {
                // Table deleted.
            }
        }

        private static void LoadSampleProducts()
        {
            Table productCatalogTable = Table.LoadTable(client, "ProductCatalog");
            // ********** Add Books *********************
            var book1 = new Document();
            book1["Id"] = 101;
            book1["Title"] = "Book 101 Title";
            book1["ISBN"] = "111-1111111111";
            book1["Authors"] = new List<string> { "Author 1" };
            book1["Price"] = -2; // *** Intentional value. Later used to illustrate
 scan.
            book1["Dimensions"] = "8.5 x 11.0 x 0.5";
            book1["PageCount"] = 500;
            book1["InPublication"] = true;
            book1["ProductCategory"] = "Book";
            productCatalogTable.PutItem(book1);

            var book2 = new Document();

            book2["Id"] = 102;
            book2["Title"] = "Book 102 Title";
            book2["ISBN"] = "222-2222222222";
            book2["Authors"] = new List<string> { "Author 1", "Author 2" }; ;
            book2["Price"] = 20;
            book2["Dimensions"] = "8.5 x 11.0 x 0.8";
            book2["PageCount"] = 600;
            book2["InPublication"] = true;
            book2["ProductCategory"] = "Book";
            productCatalogTable.PutItem(book2);

            var book3 = new Document();
            book3["Id"] = 103;
            book3["Title"] = "Book 103 Title";
            book3["ISBN"] = "333-3333333333";
```

```
            book3["Authors"] = new List<string> { "Author 1", "Author2", "Author
3" }; ;
            book3["Price"] = 2000;
            book3["Dimensions"] = "8.5 x 11.0 x 1.5";
            book3["PageCount"] = 700;
            book3["InPublication"] = false;
            book3["ProductCategory"] = "Book";
            productCatalogTable.PutItem(book3);

            // ************ Add bikes. ******************
            var bicycle1 = new Document();
            bicycle1["Id"] = 201;
            bicycle1["Title"] = "18-Bike 201"; // size, followed by some title.
            bicycle1["Description"] = "201 description";
            bicycle1["BicycleType"] = "Road";
            bicycle1["Brand"] = "Brand-Company A"; // Trek, Specialized.
            bicycle1["Price"] = 100;
            bicycle1["Color"] = new List<string> { "Red", "Black" };
            bicycle1["ProductCategory"] = "Bike";
            productCatalogTable.PutItem(bicycle1);

            var bicycle2 = new Document();
            bicycle2["Id"] = 202;
            bicycle2["Title"] = "21-Bike 202Brand-Company A";
            bicycle2["Description"] = "202 description";
            bicycle2["BicycleType"] = "Road";
            bicycle2["Brand"] = "";
            bicycle2["Price"] = 200;
            bicycle2["Color"] = new List<string> { "Green", "Black" };
            bicycle2["ProductCategory"] = "Bicycle";
            productCatalogTable.PutItem(bicycle2);

            var bicycle3 = new Document();
            bicycle3["Id"] = 203;
            bicycle3["Title"] = "19-Bike 203";
            bicycle3["Description"] = "203 description";
            bicycle3["BicycleType"] = "Road";
            bicycle3["Brand"] = "Brand-Company B";
            bicycle3["Price"] = 300;
            bicycle3["Color"] = new List<string> { "Red", "Green", "Black" };
            bicycle3["ProductCategory"] = "Bike";
            productCatalogTable.PutItem(bicycle3);

            var bicycle4 = new Document();
```

```
            bicycle4["Id"] = 204;
            bicycle4["Title"] = "18-Bike 204";
            bicycle4["Description"] = "204 description";
            bicycle4["BicycleType"] = "Mountain";
            bicycle4["Brand"] = "Brand-Company B";
            bicycle4["Price"] = 400;
            bicycle4["Color"] = new List<string> { "Red" };
            bicycle4["ProductCategory"] = "Bike";
            productCatalogTable.PutItem(bicycle4);

            var bicycle5 = new Document();
            bicycle5["Id"] = 205;
            bicycle5["Title"] = "20-Title 205";
            bicycle4["Description"] = "205 description";
            bicycle5["BicycleType"] = "Hybrid";
            bicycle5["Brand"] = "Brand-Company C";
            bicycle5["Price"] = 500;
            bicycle5["Color"] = new List<string> { "Red", "Black" };
            bicycle5["ProductCategory"] = "Bike";
            productCatalogTable.PutItem(bicycle5);
        }

        private static void LoadSampleForums()
        {
            Table forumTable = Table.LoadTable(client, "Forum");

            var forum1 = new Document();
            forum1["Name"] = "Amazon DynamoDB"; // PK
            forum1["Category"] = "Amazon Web Services";
            forum1["Threads"] = 2;
            forum1["Messages"] = 4;
            forum1["Views"] = 1000;

            forumTable.PutItem(forum1);

            var forum2 = new Document();
            forum2["Name"] = "Amazon S3"; // PK
            forum2["Category"] = "Amazon Web Services";
            forum2["Threads"] = 1;

            forumTable.PutItem(forum2);
        }

        private static void LoadSampleThreads()
```

```
        {
            Table threadTable = Table.LoadTable(client, "Thread");

            // Thread 1.
            var thread1 = new Document();
            thread1["ForumName"] = "Amazon DynamoDB"; // Hash attribute.
            thread1["Subject"] = "DynamoDB Thread 1"; // Range attribute.
            thread1["Message"] = "DynamoDB thread 1 message text";
            thread1["LastPostedBy"] = "User A";
            thread1["LastPostedDateTime"] = DateTime.UtcNow.Subtract(new TimeSpan(14,
0, 0, 0));
            thread1["Views"] = 0;
            thread1["Replies"] = 0;
            thread1["Answered"] = false;
            thread1["Tags"] = new List<string> { "index", "primarykey", "table" };

            threadTable.PutItem(thread1);

            // Thread 2.
            var thread2 = new Document();
            thread2["ForumName"] = "Amazon DynamoDB"; // Hash attribute.
            thread2["Subject"] = "DynamoDB Thread 2"; // Range attribute.
            thread2["Message"] = "DynamoDB thread 2 message text";
            thread2["LastPostedBy"] = "User A";
            thread2["LastPostedDateTime"] = DateTime.UtcNow.Subtract(new TimeSpan(21,
0, 0, 0));
            thread2["Views"] = 0;
            thread2["Replies"] = 0;
            thread2["Answered"] = false;
            thread2["Tags"] = new List<string> { "index", "primarykey", "rangekey" };

            threadTable.PutItem(thread2);

            // Thread 3.
            var thread3 = new Document();
            thread3["ForumName"] = "Amazon S3"; // Hash attribute.
            thread3["Subject"] = "S3 Thread 1"; // Range attribute.
            thread3["Message"] = "S3 thread 3 message text";
            thread3["LastPostedBy"] = "User A";
            thread3["LastPostedDateTime"] = DateTime.UtcNow.Subtract(new TimeSpan(7, 0,
0, 0));
            thread3["Views"] = 0;
            thread3["Replies"] = 0;
            thread3["Answered"] = false;
```

```
            thread3["Tags"] = new List<string> { "largeobjects", "multipart upload" };
            threadTable.PutItem(thread3);
        }

        private static void LoadSampleReplies()
        {
            Table replyTable = Table.LoadTable(client, "Reply");

            // Reply 1 - thread 1.
            var thread1Reply1 = new Document();
            thread1Reply1["Id"] = "Amazon DynamoDB#DynamoDB Thread 1"; // Hash
attribute.
            thread1Reply1["ReplyDateTime"] = DateTime.UtcNow.Subtract(new TimeSpan(21,
0, 0, 0)); // Range attribute.
            thread1Reply1["Message"] = "DynamoDB Thread 1 Reply 1 text";
            thread1Reply1["PostedBy"] = "User A";

            replyTable.PutItem(thread1Reply1);

            // Reply 2 - thread 1.
            var thread1reply2 = new Document();
            thread1reply2["Id"] = "Amazon DynamoDB#DynamoDB Thread 1"; // Hash
attribute.
            thread1reply2["ReplyDateTime"] = DateTime.UtcNow.Subtract(new TimeSpan(14,
0, 0, 0)); // Range attribute.
            thread1reply2["Message"] = "DynamoDB Thread 1 Reply 2 text";
            thread1reply2["PostedBy"] = "User B";

            replyTable.PutItem(thread1reply2);

            // Reply 3 - thread 1.
            var thread1Reply3 = new Document();
            thread1Reply3["Id"] = "Amazon DynamoDB#DynamoDB Thread 1"; // Hash
attribute.
            thread1Reply3["ReplyDateTime"] = DateTime.UtcNow.Subtract(new TimeSpan(7,
0, 0, 0)); // Range attribute.
            thread1Reply3["Message"] = "DynamoDB Thread 1 Reply 3 text";
            thread1Reply3["PostedBy"] = "User B";

            replyTable.PutItem(thread1Reply3);

            // Reply 1 - thread 2.
            var thread2Reply1 = new Document();
```

```
            thread2Reply1["Id"] = "Amazon DynamoDB#DynamoDB Thread 2"; // Hash
 attribute.
            thread2Reply1["ReplyDateTime"] = DateTime.UtcNow.Subtract(new TimeSpan(7,
 0, 0, 0)); // Range attribute.
            thread2Reply1["Message"] = "DynamoDB Thread 2 Reply 1 text";
            thread2Reply1["PostedBy"] = "User A";


            replyTable.PutItem(thread2Reply1);

            // Reply 2 - thread 2.
            var thread2Reply2 = new Document();
            thread2Reply2["Id"] = "Amazon DynamoDB#DynamoDB Thread 2"; // Hash
 attribute.
            thread2Reply2["ReplyDateTime"] = DateTime.UtcNow.Subtract(new TimeSpan(1,
 0, 0, 0)); // Range attribute.
            thread2Reply2["Message"] = "DynamoDB Thread 2 Reply 2 text";
            thread2Reply2["PostedBy"] = "User A";

            replyTable.PutItem(thread2Reply2);
        }
    }
}
```

DynamoDB example application using the AWS SDK for Python (Boto): Tic-tac-toe

The Tic-Tac-Toe game is an example web application built on Amazon DynamoDB. The application uses the AWS SDK for Python (Boto) to make the necessary DynamoDB calls to store game data in a DynamoDB table, and the Python web framework Flask to illustrate end-to-end application development in DynamoDB, including how to model data. It also demonstrates best practices when it comes to modeling data in DynamoDB, including the table you create for the game application, the primary key you define, additional indexes you need based on your query requirements, and the use of concatenated value attributes.

You play the Tic-Tac-Toe application on the web as follows:

1. You log in to the application home page.
2. You then invite another user to play the game as your opponent.

Until another user accepts your invitation, the game status remains as `PENDING`. After an opponent accepts the invite, the game status changes to `IN_PROGRESS`.

3. The game begins after the opponent logs in and accepts the invite.
4. The application stores all game moves and status information in a DynamoDB table.
5. The game ends with a win or a draw, which sets the game status to `FINISHED`.

The end-to-end application building exercise is described in steps:

- **Step 1: Deploy and test locally** – In this section, you download, deploy, and test the application on your local computer. You will create the required tables in the downloadable version of DynamoDB.
- **Step 2: Examine the data model and implementation details** – This section first describes in detail the data model, including the indexes and the use of the concatenated value attribute. Then the section explains how the application works.
- **Step 3: Deploy in production using the DynamoDB service** – This section focuses on deployment considerations in production. In this step, you create a table using the Amazon DynamoDB service and deploy the application using AWS Elastic Beanstalk. When you have the application in production, you also grant appropriate permissions so the application can access the DynamoDB table. The instructions in this section walk you through the end-to-end production deployment.
- **Step 4: Clean up resources** – This section highlights areas that are not covered by this example. The section also provides steps for you to remove the AWS resources you created in the preceding steps so that you avoid incurring any charges.

Step 1: Deploy and test locally

Topics

- 1.1: Download and install the required packages
- 1.2: Test the game application

In this step you download, deploy, and test the Tic-Tac-Toe game application on your local computer. Instead of using the Amazon DynamoDB web service, you will download DynamoDB to your computer, and create the required table there.

1.1: Download and install the required packages

You will need the following to test this application locally:

- Python
- Flask (a microframework for Python)
- AWS SDK for Python (Boto)
- DynamoDB running on your computer
- Git

To get these tools, do the following:

1. Install Python. For step-by-step instructions, see Download Python.

   The Tic-Tac-Toe application has been tested using Python version 2.7.
2. Install Flask and AWS SDK for Python (Boto) using the Python Package Installer (PIP):
   - Install PIP.

     For instructions, see Install PIP. On the installation page, choose the **get-pip.py** link, and then save the file. Then open a command terminal as an administrator, and enter the following at the command prompt.

     ```
     python.exe get-pip.py
     ```

     On Linux, you don't specify the `.exe` extension. You only specify `python get-pip.py`.
   - Using PIP, install the Flask and Boto packages using the following code.

     ```
     pip install Flask
     pip install boto
     pip install configparser
     ```
3. Download DynamoDB to your computer. For instructions on how to run it, see Setting up DynamoDB local (downloadable version).
4. Download the Tic-Tac-Toe application:
   a. Install Git. For instructions, see git downloads.
   b. Run the following code to download the application.

```
git clone https://github.com/awslabs/dynamodb-tictactoe-example-app.git
```

1.2: Test the game application

To test the Tic-Tac-Toe application, you need to run DynamoDB locally on your computer.

To run the tic-tac-toe application

1. Start DynamoDB.
2. Start the web server for the Tic-Tac-Toe application.

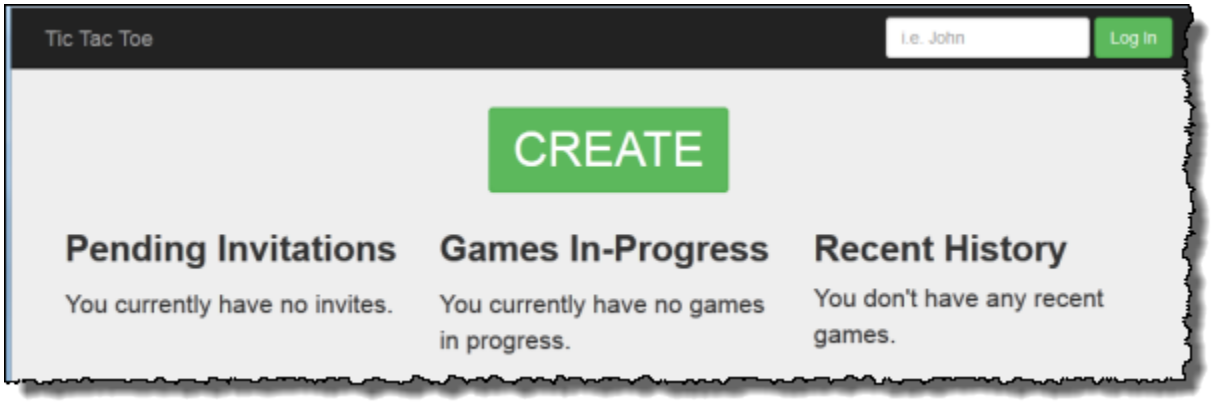
   To do so, open a command terminal, navigate to the folder where you downloaded the Tic-Tac-Toe application, and run the application locally using the following code.

   ```
   python.exe application.py --mode local --serverPort 5000 --port 8000
   ```

   On Linux, you don't specify the `.exe` extension.
3. Open your web browser, and enter the following.

   ```
   http://localhost:5000/
   ```

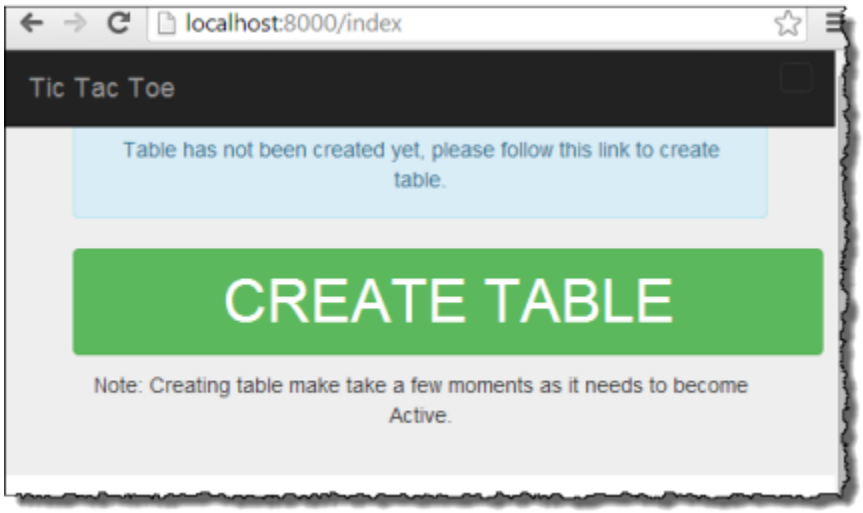
   The browser shows the home page.



4. Enter **`user1`** in the **Log in** box to log in as user1.

> **Note**
>
> This example application does not perform any user authentication. The user ID is only used to identify players. If two players log in with the same alias, the application works as if you are playing in two different browsers.

5. If this is your first time playing the game, a page appears requesting you to create the required table (`Games`) in DynamoDB. Choose **CREATE TABLE**.



6. Choose **CREATE** to create the first tic-tac-toe game.
7. Enter **`user2`** in the **Choose an Opponent** box, and choose **Create Game!**

   Doing this creates the game by adding an item in the `Games` table. It sets the game status to `PENDING`.

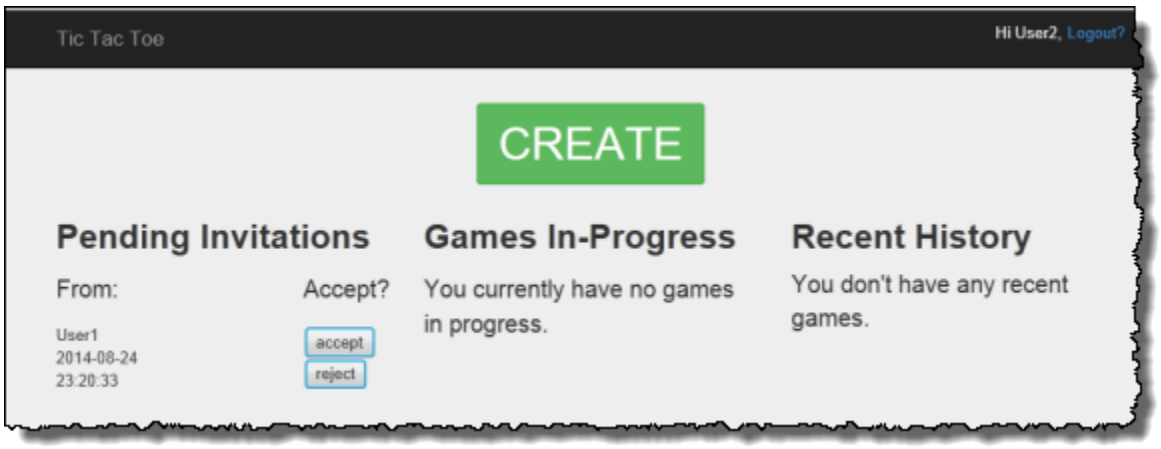
8. Open another browser window, and enter the following.

```
http://localhost:5000/
```

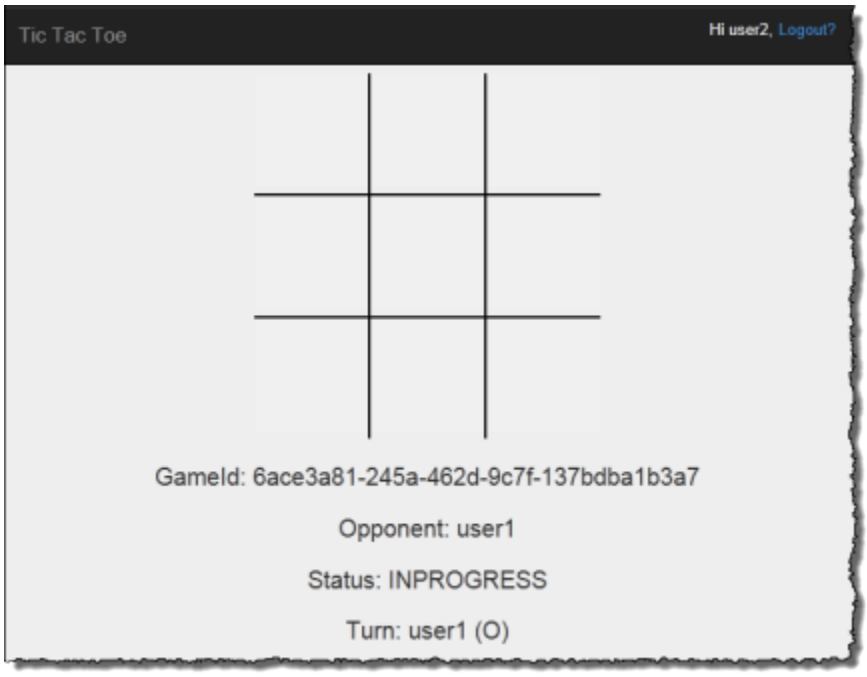
The browser passes information through cookies, so you should use incognito mode or private browsing so that your cookies don't carry over.

9. Log in as user2.

   A page appears that shows a pending invitation from user1.



10. Choose **accept** to accept the invitation.



   The game page appears with an empty tic-tac-toe grid. The page also shows relevant game information such as the game ID, whose turn it is, and game status.

11. Play the game.

For each user move, the web service sends a request to DynamoDB to conditionally update the game item in the `Games` table. For example, the conditions ensure that the move is valid, the square that the user chose is available, and that it was the turn of the user who made the move. For a valid move, the update operation adds a new attribute corresponding to the selection on the board. The update operation also sets the value of the existing attribute to the user who can make the next move.

On the game page, the application makes asynchronous JavaScript calls every second, for up to 5 minutes, to check if the game state in DynamoDB has changed. If it has, the application updates the page with new information. After 5 minutes, the application stops making the requests, and you need to refresh the page to get updated information.

Step 2: Examine the data model and implementation details

Topics

- 2.1: Basic data model
- 2.2: Application in action (code walkthrough)

2.1: Basic data model

This example application highlights the following DynamoDB data model concepts:

- **Table** – In DynamoDB, a table is a collection of items (that is, records), and each item is a collection of name-value pairs called attributes.

  In this Tic-Tac-Toe example, the application stores all game data in a table, `Games`. The application creates one item in the table per game and stores all game data as attributes. A tic-tac-toe game can have up to nine moves. Because DynamoDB tables do not have a schema in cases where only the primary key is the required attribute, the application can store varying number of attributes per game item.

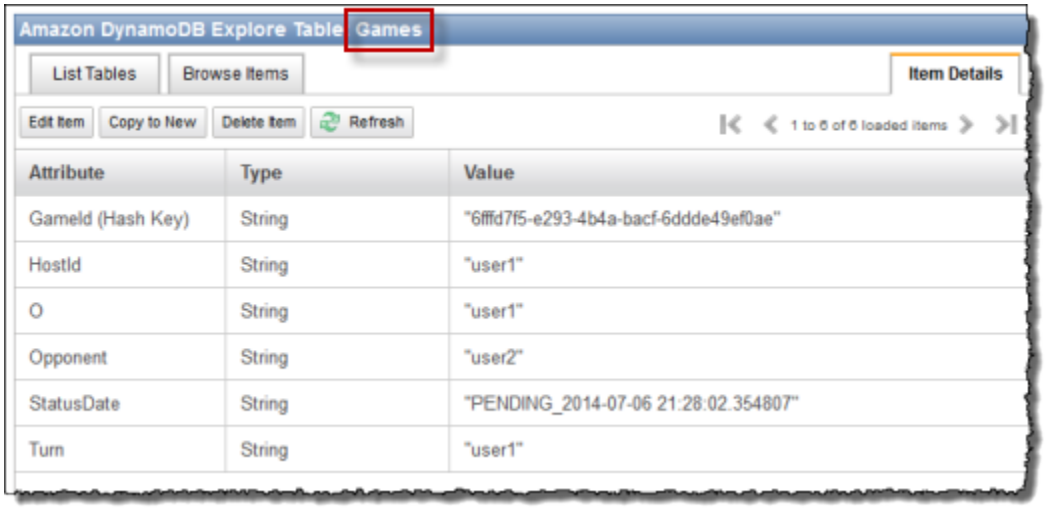
  The `Games` table has a simple primary key made of one attribute, `GameId`, of string type. The application assigns a unique ID to each game. For more information on DynamoDB primary keys, see Primary key.

  When a user initiates a tic-tac-toe game by inviting another user to play, the application creates a new item in the `Games` table with attributes storing game metadata, such as the following:

  - `HostId`, the user who initiated the game.

- `Opponent`, the user who was invited to play.
- The user whose turn it is to play. The user who initiated the game plays first.
- The user who uses the **O** symbol on the board. The user who initiates the games uses the **O** symbol.

In addition, the application creates a `StatusDate` concatenated attribute, marking the initial game state as `PENDING`. The following screenshot shows an example item as it appears in the DynamoDB console:

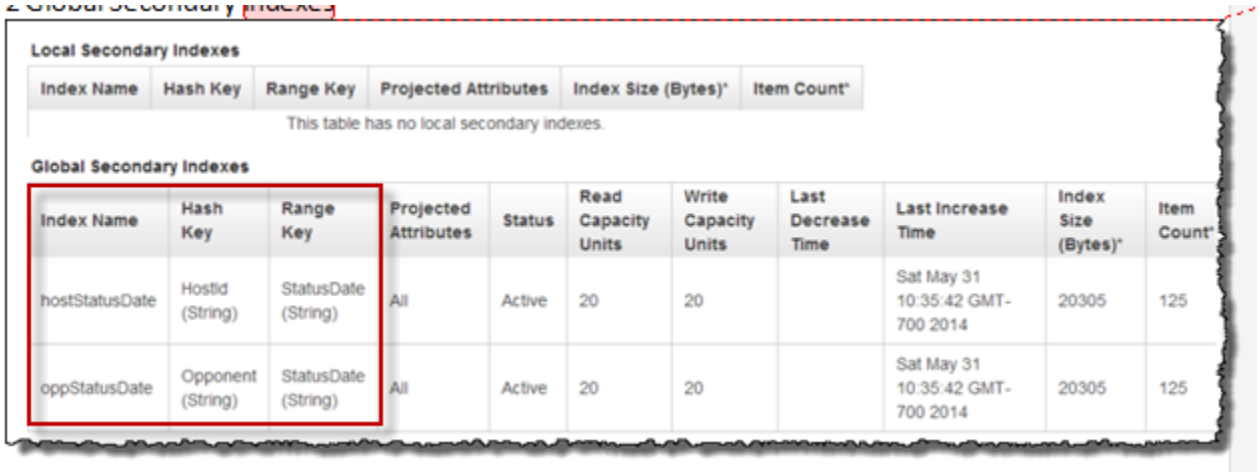


As the game progresses, the application adds one attribute to the table for each game move. The attribute name is the board position, for example `TopLeft` or `BottomRight`. For example, a move might have a `TopLeft` attribute with the value 0, a `TopRight` attribute with the value 0, and a `BottomRight` attribute with the value X. The attribute value is either 0 or X, depending on which user made the move. For example, consider the following board.

- **Concatenated value attributes** – The `StatusDate` attribute illustrates a concatenated value attribute. In this approach, instead of creating separate attributes to store game status (`PENDING`, `IN_PROGRESS`, and `FINISHED`) and date (when the last move was made), you combine them as single attribute, for example `IN_PROGRESS_2014-04-30 10:20:32`.

  The application then uses the `StatusDate` attribute in creating secondary indexes by specifying `StatusDate` as a sort key for the index. The benefit of using the `StatusDate` concatenated value attribute is further illustrated in the indexes discussed next.

- **Global secondary indexes** – You can use the table's primary key, `GameId`, to efficiently query the table to find a game item. To query the table on attributes other than the primary key attributes, DynamoDB supports the creation of secondary indexes. In this example application, you build the following two secondary indexes:



Local Secondary Indexes

Index Name	Hash Key	Range Key	Projected Attributes	Index Size (Bytes)*	Item Count*

This table has no local secondary indexes.

Global Secondary Indexes

Index Name	Hash Key	Range Key	Projected Attributes	Status	Read Capacity Units	Write Capacity Units	Last Decrease Time	Last Increase Time	Index Size (Bytes)*	Item Count*
hostStatusDate	HostId (String)	StatusDate (String)	All	Active	20	20		Sat May 31 10:35:42 GMT-700 2014	20305	125
oppStatusDate	Opponent (String)	StatusDate (String)	All	Active	20	20		Sat May 31 10:35:42 GMT-700 2014	20305	125

  - **HostId-StatusDate-index**. This index has `HostId` as a partition key and `StatusDate` as a sort key. You can use this index to query on `HostId`, for example to find games hosted by a particular user.
  - **OpponentId-StatusDate-index**. This index has `OpponentId` as a partition key and `StatusDate` as a sort key. You can use this index to query on `Opponent`, for example to find games where a particular user is the opponent.

  These indexes are called global secondary indexes because the partition key in these indexes is not the same the partition key (`GameId`), used in the primary key of the table.

  Note that both the indexes specify `StatusDate` as a sort key. Doing this enables the following:

  - You can query using the `BEGINS_WITH` comparison operator. For example, you can find all games with the `IN_PROGRESS` attribute hosted by a particular user. In this case, the `BEGINS_WITH` operator checks for the `StatusDate` value that begins with `IN_PROGRESS`.

- DynamoDB stores the items in the index in sorted order, by sort key value. So if all status prefixes are the same (for example, `IN_PROGRESS`), the ISO format used for the date part will have items sorted from oldest to the newest. This approach enables certain queries to be performed efficiently, for example the following:
  - Retrieve up to 10 of the most recent `IN_PROGRESS` games hosted by the user who is logged in. For this query, you specify the `HostId-StatusDate-index` index.
  - Retrieve up to 10 of the most recent `IN_PROGRESS` games where the user logged in is the opponent. For this query, you specify the `OpponentId-StatusDate-index` index.

For more information about secondary indexes, see Improving data access with secondary indexes in DynamoDB.

2.2: Application in action (code walkthrough)

This application has two main pages:

- **Home page** – This page provides the user a simple login, a **CREATE** button to create a new tic-tac-toe game, a list of games in progress, game history, and any active pending game invitations.

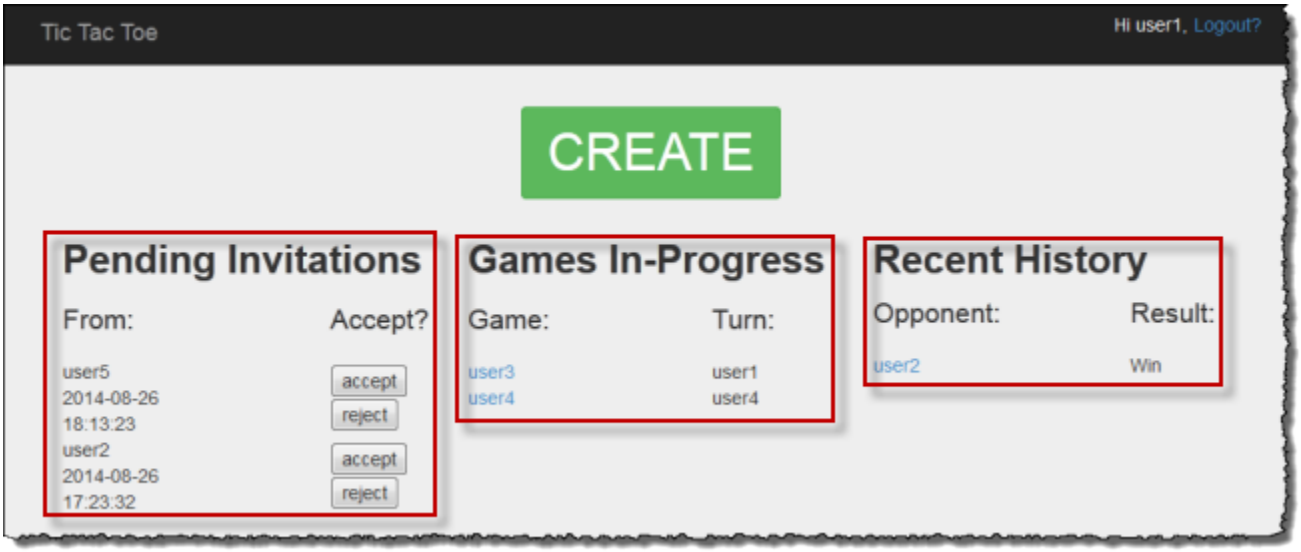
  The home page is not refreshed automatically; you must refresh the page to refresh the lists.
- **Game page** – This page shows the tic-tac-toe grid where users play.

  The application updates the game page automatically every second. The JavaScript in your browser calls the Python web server every second to query the Games table whether the game items in the table have changed. If they have, JavaScript triggers a page refresh so that the user sees the updated board.

Let us see in detail how the application works.

Home page

After the user logs in, the application displays the following three lists of information.



- **Invitations** – This list shows up to the 10 most recent invitations from others that are pending acceptance by the user who is logged in. In the preceding screenshot, user1 has invitations from user5 and user2 pending.
- **Games in-progress** – This list shows up to the 10 most recent games that are in progress. These are games that the user is actively playing, which have the status `IN_PROGRESS`. In the screenshot, user1 is actively playing a tic-tac-toe game with user3 and user4.
- **Recent history** – This list shows up to the 10 most recent games that the user finished, which have the status `FINISHED`. In game shown in the screenshot, user1 has previously played with user2. For each completed game, the list shows the game result.

In the code, the `index` function (in `application.py`) makes the following three calls to retrieve game status information:

```
inviteGames     = controller.getGameInvites(session["username"])
inProgressGames = controller.getGamesWithStatus(session["username"], "IN_PROGRESS")
finishedGames   = controller.getGamesWithStatus(session["username"], "FINISHED")
```

Each of these calls returns a list of items from DynamoDB that are wrapped by the `Game` objects. It is easy to extract data from these objects in the view. The index function passes these object lists to the view to render the HTML.

```
return render_template("index.html",
                       user=session["username"],
                       invites=inviteGames,
```

```
                        inprogress=inProgressGames,
                        finished=finishedGames)
```

The Tic-Tac-Toe application defines the `Game` class primarily to store game data retrieved from DynamoDB. These functions return lists of `Game` objects that enable you to isolate the rest of the application from code related to Amazon DynamoDB items. Thus, these functions help you decouple your application code from the details of the data store layer.

The architectural pattern described here is also referred as the model-view-controller (MVC) UI pattern. In this case, the `Game` object instances (representing data) are the model, and the HTML page is the view. The controller is divided into two files. The `application.py` file has the controller logic for the Flask framework, and the business logic is isolated in the `gameController.py` file. That is, the application stores everything that has to do with DynamoDB SDK in its own separate file in the `dynamodb` folder.

Let us review the three functions and how they query the Games table using global secondary indexes to retrieve relevant data.

Using getGameInvites to get the list of pending game invitations

The `getGameInvites` function retrieves the list of the 10 most recent pending invitations. These games have been created by users, but the opponents have not accepted the game invitations. For these games, the status remains `PENDING` until the opponent accepts the invite. If the opponent declines the invite, the application removes the corresponding item from the table.

The function specifies the query as follows:

- It specifies the `OpponentId-StatusDate-index` index to use with the following index key values and comparison operators:
  - The partition key is `OpponentId` and takes the index key *`user ID`*.
  - The sort key is `StatusDate` and takes the comparison operator and index key value `beginswith="PENDING_"`.

  You use the `OpponentId-StatusDate-index` index to retrieve games to which the logged-in user is invited—that is, where the logged-in user is the opponent.
- The query limits the result to 10 items.

```
gameInvitesIndex = self.cm.getGamesTable().query(
```

```
                                        Opponent__eq=user,
                                        StatusDate__beginswith="PENDING_",
                                        index="OpponentId-StatusDate-index",
                                        limit=10)
```

In the index, for each `OpponentId` (the partition key) DynamoDB keeps items sorted by `StatusDate` (the sort key). Therefore, the games that the query returns will be the 10 most recent games.

Using getGamesWithStatus to get the list of games with a specific status

After an opponent accepts a game invitation, the game status changes to `IN_PROGRESS`. After the game completes, the status changes to `FINISHED`.

Queries to find games that are either in progress or finished are the same except for the different status value. Therefore, the application defines the `getGamesWithStatus` function, which takes the status value as a parameter.

```
inProgressGames = controller.getGamesWithStatus(session["username"], "IN_PROGRESS")
finishedGames   = controller.getGamesWithStatus(session["username"], "FINISHED")
```

The following section discusses in-progress games, but the same description also applies to finished games.

A list of in-progress games for a given user includes both the following:

- In-progress games hosted by the user
- In-progress games where the user is the opponent

The `getGamesWithStatus` function runs the following two queries, each time using the appropriate secondary index.

- The function queries the `Games` table using the `HostId-StatusDate-index` index. For the index, the query specifies primary key values—both the partition key (`HostId`) and sort key (`StatusDate`) values, along with comparison operators.

  ```
  hostGamesInProgress = self.cm.getGamesTable ().query(HostId__eq=user,
                                                       StatusDate__beginswith=status,
                                                       index="HostId-StatusDate-index",
                                                       limit=10)
  ```

Note the Python syntax for comparison operators:

- `HostId__eq=user` specifies the equality comparison operator.
- `StatusDate__beginswith=status` specifies the `BEGINS_WITH` comparison operator.

- The function queries the `Games` table using the `OpponentId-StatusDate-index` index.

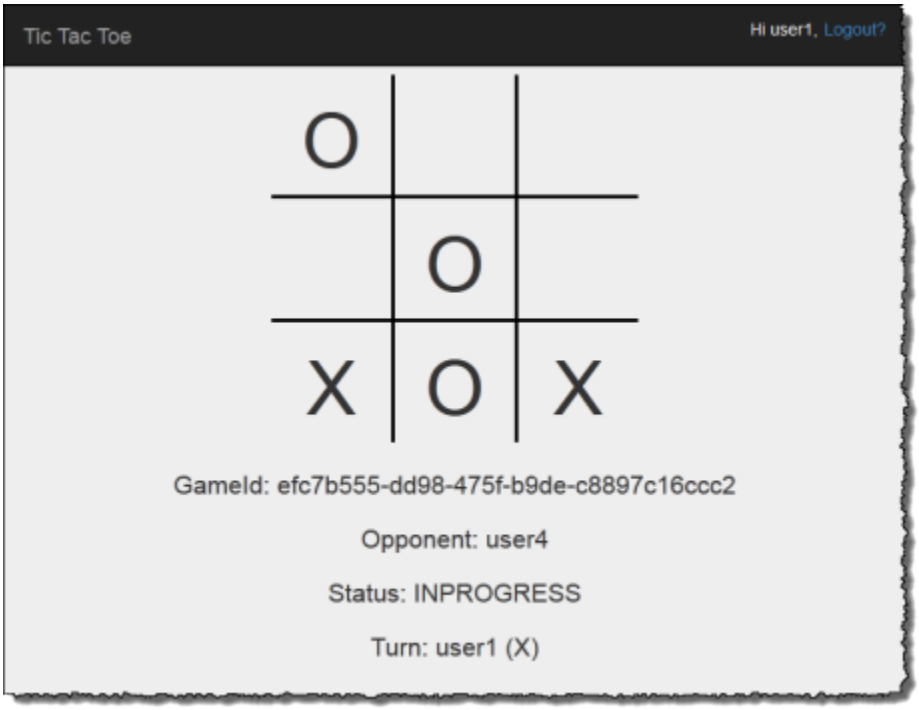
```
oppGamesInProgress = self.cm.getGamesTable().query(Opponent__eq=user,
                                                   StatusDate__beginswith=status,
                                                   index="OpponentId-StatusDate-index",
                                                   limit=10)
```

- The function then combines the two lists, sorts, and for the first 0 to 10 items creates a list of the `Game` objects and returns the list to the calling function (that is, the index).

```
games = self.mergeQueries(hostGamesInProgress,
                          oppGamesInProgress)
return games
```

Game page

The game page is where the user plays tic-tac-toe games. It shows the game grid along with game-relevant information. The following screenshot shows an example game in progress:



The application displays the game page in the following situations:

- The user creates a game inviting another user to play.

  In this case, the page shows the user as host and the game status as `PENDING`, waiting for the opponent to accept.
- The user accepts one of the pending invitations on the home page.

  In this case, the page shows the user as the opponent and game status as `IN_PROGRESS`.

A user selection on the board generates a form `POST` request to the application. That is, Flask calls the `selectSquare` function (in `application.py`) with the HTML form data. This function, in turn, calls the `updateBoardAndTurn` function (in `gameController.py`) to update the game item as follows:

- It adds a new attribute specific to the move.
- It updates the `Turn` attribute value to the user whose turn is next.

```
controller.updateBoardAndTurn(item, value, session["username"])
```

The function returns true if the item update was successful; otherwise, it returns false. Note the following about the `updateBoardAndTurn` function:

- The function calls the `update_item` function of the SDK for Python to make a finite set of updates to an existing item. The function maps to the `UpdateItem` operation in DynamoDB. For more information, see UpdateItem.

  > **Note**
  >
  > The difference between the `UpdateItem` and `PutItem` operations is that `PutItem` replaces the entire item. For more information, see PutItem.

For the `update_item` call, the code identifies the following:

- The primary key of the `Games` table (that is, `ItemId`).

```
key = { "GameId" : { "S" : gameId } }
```

- The new attribute to add, specific to the current user move, and its value (for example, `TopLeft="X"`).

```
attributeUpdates = {
    position : {
        "Action" : "PUT",
        "Value" : { "S" : representation }
    }
}
```

- Conditions that must be true for the update to take place:
  - The game must be in progress. That is, the `StatusDate` attribute value must begin with `IN_PROGRESS`.
  - The current turn must be a valid user turn as specified by the `Turn` attribute.
  - The square that the user chose must be available. That is, the attribute corresponding to the square must not exist.

```
expectations = {"StatusDate" : {"AttributeValueList": [{"S" : "IN_PROGRESS_"}],
    "ComparisonOperator": "BEGINS_WITH"},
    "Turn" : {"Value" : {"S" : current_player}},
    position : {"Exists" : False}}
```

Now the function calls `update_item` to update the item.

```
self.cm.db.update_item("Games", key=key,
    attribute_updates=attributeUpdates,
    expected=expectations)
```

After the function returns, the `selectSquare` function calls redirect, as shown in the following example.

```
redirect("/game="+gameId)
```

This call causes the browser to refresh. As part of this refresh, the application checks to see if the game has ended in a win or draw. If it has, the application updates the game item accordingly.

Step 3: Deploy in production using the DynamoDB service

Topics

- 3.1: Create an IAM role for Amazon EC2
- 3.2: Create the games table in Amazon DynamoDB
- 3.3: Bundle and deploy the tic-tac-toe application code
- 3.4: Set up the AWS Elastic Beanstalk environment

In the preceding sections, you deployed and tested the Tic-Tac-Toe application locally on your computer using DynamoDB local. Now, you deploy the application in production as follows:

- Deploy the application using AWS Elastic Beanstalk, an easy-to-use service for deploying and scaling web applications and services. For more information, see Deploying a flask application to AWS Elastic Beanstalk.

  Elastic Beanstalk launches one or more Amazon Elastic Compute Cloud (Amazon EC2) instances, which you configure through Elastic Beanstalk, on which your Tic-Tac-Toe application will run.
- Using the Amazon DynamoDB service, create a `Games` table that exists on AWS rather than locally on your computer.

In addition, you also have to configure permissions. Any AWS resources you create, such as the `Games` table in DynamoDB, are private by default. Only the resource owner, that is the AWS account that created the `Games` table, can access this table. Thus, by default your Tic-Tac-Toe application cannot update the `Games` table.

To grant necessary permissions, you create an AWS Identity and Access Management (IAM) role and grant this role permissions to access the `Games` table. Your Amazon EC2 instance first assumes this role. In response, AWS returns temporary security credentials that the Amazon EC2 instance can use to update the `Games` table on behalf of the Tic-Tac-Toe application. When you configure your Elastic Beanstalk application, you specify the IAM role that the Amazon EC2 instance or instances can assume. For more information about IAM roles, see IAM roles for amazon EC2 in the *Amazon EC2 User Guide.*

> **Note**
>
> Before you create Amazon EC2 instances for the Tic-Tac-Toe application, you must first decide the AWS Region where you want Elastic Beanstalk to create the instances. After you create the Elastic Beanstalk application, you provide the same Region name and endpoint in a configuration file. The Tic-Tac-Toe application uses information in this file to create the `Games` table and send subsequent requests in a specific AWS Region. Both the DynamoDB `Games` table and the Amazon EC2 instances that Elastic Beanstalk launches must be in the same Region. For a list of available Regions, see Amazon DynamoDB in the *Amazon Web Services General Reference*.

In summary, you do the following to deploy the Tic-Tac-Toe application in production:

1. Create an IAM role using the IAM service. You attach a policy to this role granting permissions for DynamoDB actions to access the `Games` table.
2. Bundle the Tic-Tac-Toe application code and a configuration file, and create a `.zip` file. You use this `.zip` file to give the Tic-Tac-Toe application code to Elastic Beanstalk to put on your servers. For more information about creating a bundle, see Creating an application source bundle in the *AWS Elastic Beanstalk Developer Guide*.

   In the configuration file (`beanstalk.config`), you provide AWS Region and endpoint information. The Tic-Tac-Toe application uses this information to determine which DynamoDB Region to talk to.
3. Set up the Elastic Beanstalk environment. Elastic Beanstalk launches an Amazon EC2 instance or instances and deploys your Tic-Tac-Toe application bundle on them. After the Elastic Beanstalk environment is ready, you provide the configuration file name by adding the `CONFIG_FILE` environment variable.
4. Create the DynamoDB table. Using the Amazon DynamoDB service, you create the `Games` table on AWS, rather than locally on your computer. Remember, this table has a simple primary key made of the `GameId` partition key of string type.
5. Test the game in production.

3.1: Create an IAM role for Amazon EC2

Creating an IAM role of the **Amazon EC2** type allows the Amazon EC2 instance that is running your Tic-Tac-Toe application to assume the correct role and make application requests to access the `Games` table. When creating the role, choose the **Custom Policy** option and copy and paste the following policy.

JSON

```
{
   "Version":"2012-10-17",
   "Statement":[
      {
         "Action":[
            "dynamodb:ListTables"
         ],
         "Effect":"Allow",
         "Resource":"*"
      },
      {
         "Action":[
            "dynamodb:*"
         ],
         "Effect":"Allow",
         "Resource":[
            "arn:aws:dynamodb:us-west-2:922852403271:table/Games",
            "arn:aws:dynamodb:us-west-2:922852403271:table/Games/index/*"
         ]
      }
   ]
}
```

For further instructions, see Creating a role for an AWS service (AWS Management Console) in the *IAM User Guide*.

3.2: Create the games table in Amazon DynamoDB

The `Games` table in DynamoDB stores game data. If the table does not exist, the application creates the table for you. In this case, let the application create the `Games` table.

3.3: Bundle and deploy the tic-tac-toe application code

If you followed this example's steps, then you already have the downloaded the Tic-Tac-Toe application. If not, download the application and extract all the files to a folder on your local computer. For instructions, see Step 1: Deploy and test locally.

After you extract all files, you will have a `code` folder. To hand off this folder to Elastic Beanstalk, you bundle the contents of this folder as a `.zip` file. First, you add a configuration file to that folder. Your application uses the Region and endpoint information to create a DynamoDB table in the specified Region and make subsequent table operation requests using the specified endpoint.

1. Switch to the folder where you downloaded the Tic-Tac-Toe application.
2. In the root folder of the application, create a text file named `beanstalk.config` with the following content.

   ```
   [dynamodb]
   region=<AWS region>
   endpoint=<DynamoDB endpoint>
   ```

   For example, you might use the following content.

   ```
   [dynamodb]
   region=us-west-2
   endpoint=dynamodb.us-west-2.amazonaws.com
   ```

   For a list of available Regions, see Amazon DynamoDB in the *Amazon Web Services General Reference.*

   > **Important**
   >
   > The Region specified in the configuration file is the location where the Tic-Tac-Toe application creates the `Games` table in DynamoDB. You must create the Elastic Beanstalk application discussed in the next section in the same Region.

> **Note**
>
> When you create your Elastic Beanstalk application, you request to launch an environment where you can choose the environment type. To test the Tic-Tac-Toe example application, you can choose the **Single Instance** environment type, skip the following, and go to the next step.
> However, the **Load balancing, autoscaling** environment type provides a highly available and scalable environment, something you should consider when you create and deploy other applications. If you choose this environment type, you also need to generate a UUID and add it to the configuration file, as shown following.
>
> ```
> [dynamodb]
> region=us-west-2
> endpoint=dynamodb.us-west-2.amazonaws.com
> [flask]
> secret_key= 284e784d-1a25-4a19-92bf-8eeb7a9example
> ```
>
> In client-server communication, when the server sends a response, for security's sake the server sends a signed cookie that the client sends back to the server in the next request. When there is only one server, the server can locally generate an encryption key when it starts. When there are many servers, they all need to know the same encryption key; otherwise, they won't be able to read cookies set by the peer servers. By adding `secret_key` to the configuration file, you tell all servers to use this encryption key.

3. Zip the content of the root folder of the application (which includes the `beanstalk.config` file)—for example, `TicTacToe.zip`.
4. Upload the `.zip` file to an Amazon Simple Storage Service (Amazon S3) bucket. In the next section, you provide this `.zip` file to Elastic Beanstalk to upload on the server or servers.

   For instructions on how to upload to an Amazon S3 bucket, see Create a bucket and Add an object to a bucket in the *Amazon Simple Storage Service User Guide.*

3.4: Set up the AWS Elastic Beanstalk environment

In this step, you create an Elastic Beanstalk application, which is a collection of components including environments. For this example, you launch one Amazon EC2 instance to deploy and run your Tic-Tac-Toe application.

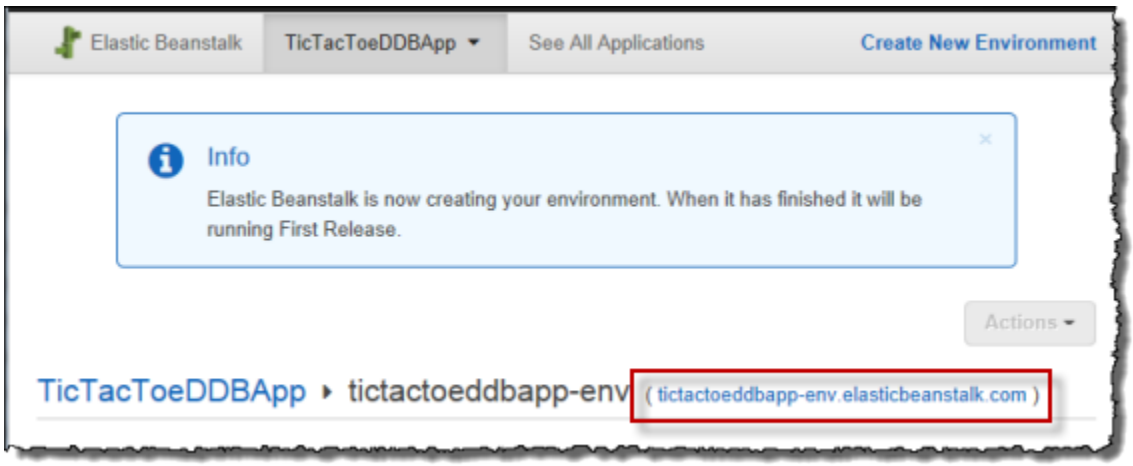
1. Enter the following custom URL to set up an Elastic Beanstalk console to set up the environment.

   ```
   https://console.aws.amazon.com/elasticbeanstalk/?region=<AWS-Region>#/
   newApplication
   ?applicationName=TicTacToeyour-name
   &solutionStackName=Python
   &sourceBundleUrl=https://s3.amazonaws.com/<bucket-name>/TicTacToe.zip
   &environmentType=SingleInstance
   &instanceType=t1.micro
   ```

   For more information about custom URLs, see Constructing a Launch Now URL in the *AWS Elastic Beanstalk Developer Guide.* For the URL, note the following:

   - You must provide an AWS Region name (the same as the one you provided in the configuration file), an Amazon S3 bucket name, and the object name.
   - For testing, the URL requests the **SingleInstance** environment type, and `t1.micro` as the instance type.
   - The application name must be unique. Thus, in the preceding URL, we suggest you prepend your name to the `applicationName`.

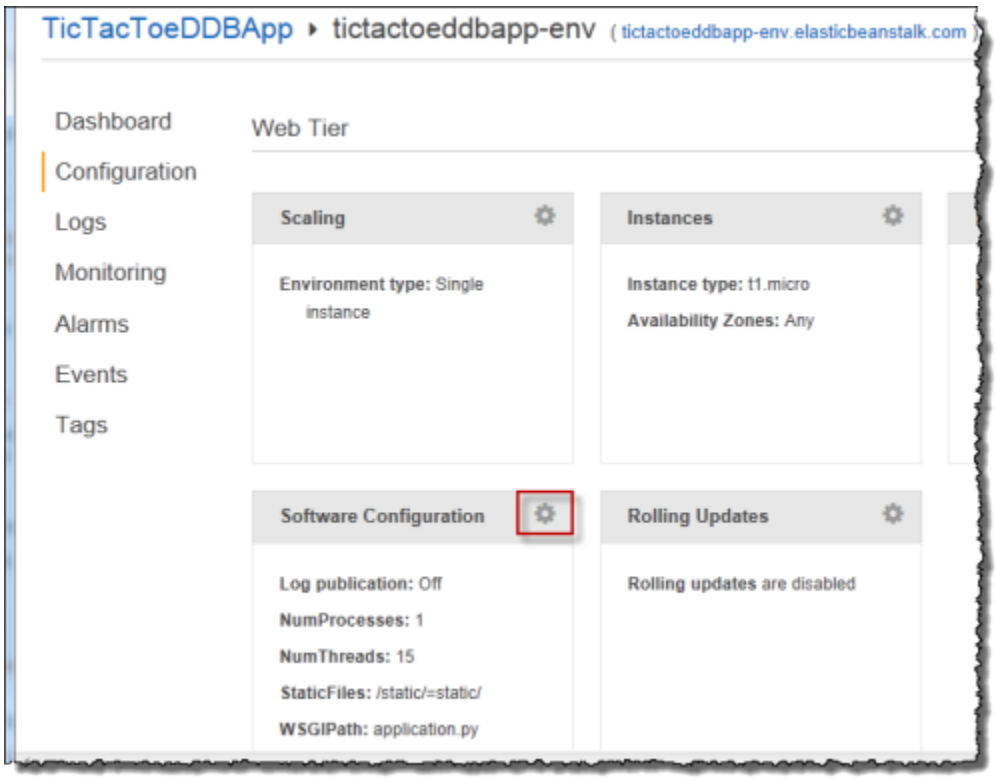
   Doing this opens the Elastic Beanstalk console. In some cases, you might need to sign in.
2. In the Elastic Beanstalk console, choose **Review and Launch**, and then choose **Launch**.
3. Note the URL for future reference. This URL opens your Tic-Tac-Toe application home page.



4. Configure the Tic-Tac-Toe application so it knows the location of the configuration file.

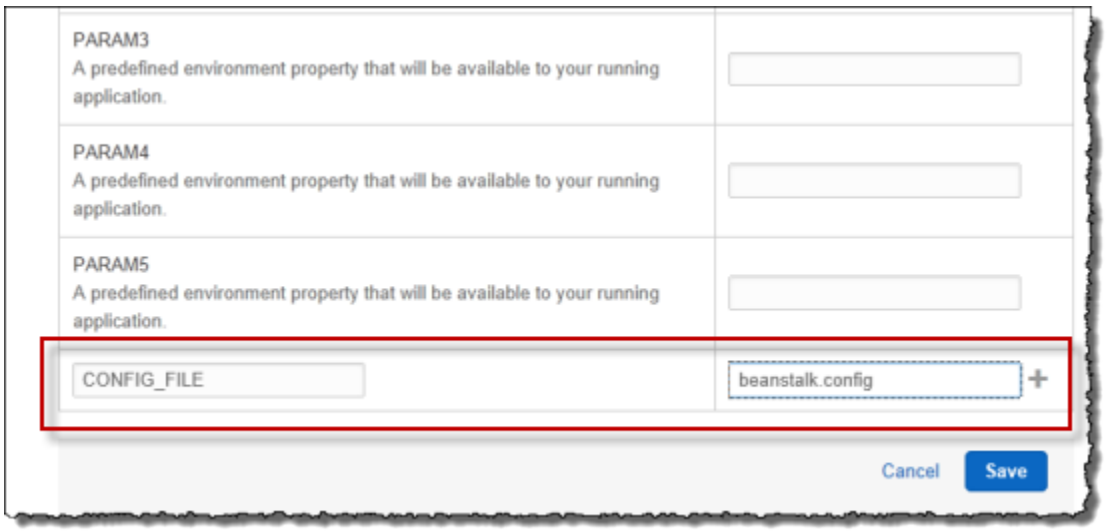
   After Elastic Beanstalk creates the application, choose **Configuration**.

   a. Choose the gear icon next to **Software Configuration**, as shown in the following screenshot.



   b. At the end of the **Environment Properties** section, enter **`CONFIG_FILE`** and its value **`beanstalk.config`**, and then choose **Save**.

      It might take a few minutes for this environment update to complete.

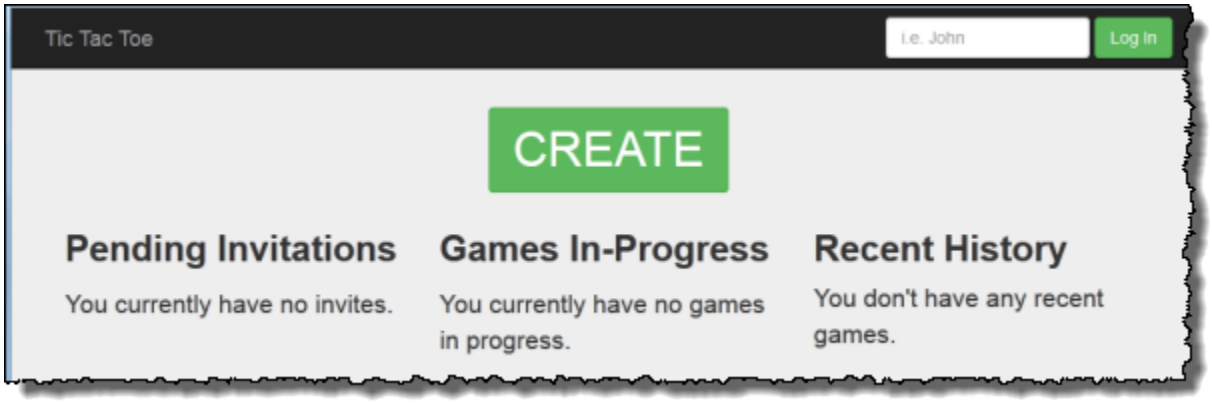


After the update completes, you can play the game.

5. In the browser, enter the URL you copied in the previous step, as shown in the following example.

```
http://<pen-name>.elasticbeanstalk.com
```

Doing this opens the application home page.



6. Log in as testuser1, and choose **CREATE** to start a new tic-tac-toe game.
7. Enter **`testuser2`** in the **Choose an Opponent** box.

8. Open another browser window.

   Make sure that you clear all cookies in your browser window so you won't be logged in as same user.

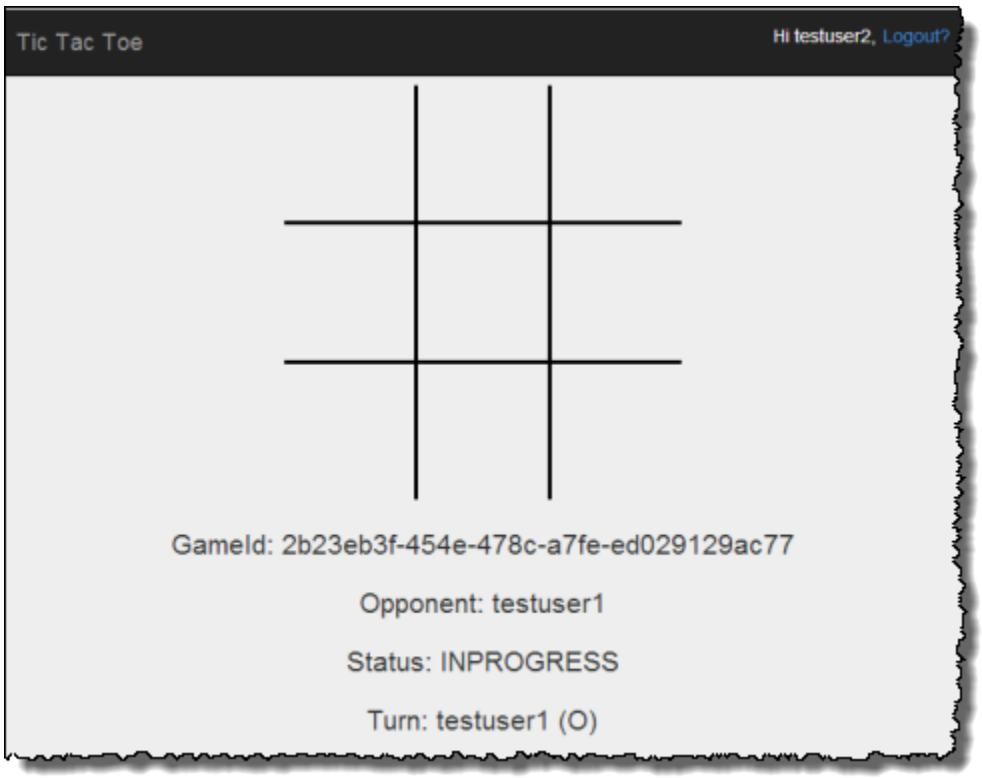
9. Enter the same URL to open the application home page, as shown in the following example.

   ```
   http://<env-name>.elasticbeanstalk.com
   ```

10. Log in as testuser2.

11. For the invitation from testuser1 in the list of pending invitations, choose **accept**.

12. Now the game page appears.



Both testuser1 and testuser2 can play the game. For each move, the application saves the move in the corresponding item in the `Games` table.

Step 4: Clean up resources

Now you have completed the Tic-Tac-Toe application deployment and testing. The application covers end-to-end web application development on Amazon DynamoDB, except for user authentication. The application uses the login information on the home page only to add a player name when creating a game. In a production application, you would add the necessary code to perform user login and authentication.

If you are done testing, you can remove the resources you created to test the Tic-Tac-Toe application to avoid incurring any charges.

To remove the resources you created

1. Remove the `Games` table that you created in DynamoDB.
2. Terminate the Elastic Beanstalk environment to free up the Amazon EC2 instances.
3. Delete the IAM role that you created.

4. Remove the object that you created in Amazon S3.

Reserved words in DynamoDB

The following keywords are reserved for use by DynamoDB. Don't use any of these words as attribute names in expressions. This list isn't case-sensitive.

If you need to write an expression containing an attribute name that conflicts with a DynamoDB reserved word, you can define an expression attribute name to use in the place of the reserved word. For more information, see Expression attribute names (aliases) in DynamoDB.

```
ABORT
ABSOLUTE
ACTION
ADD
AFTER
AGENT
AGGREGATE
ALL
ALLOCATE
ALTER
ANALYZE
AND
ANY
ARCHIVE
ARE
ARRAY
AS
ASC
ASCII
ASENSITIVE
ASSERTION
ASYMMETRIC
AT
ATOMIC
ATTACH
ATTRIBUTE
AUTH
AUTHORIZATION
AUTHORIZE
AUTO
```

```
AVG
BACK
BACKUP
BASE
BATCH
BEFORE
BEGIN
BETWEEN
BIGINT
BINARY
BIT
BLOB
BLOCK
BOOLEAN
BOTH
BREADTH
BUCKET
BULK
BY
BYTE
CALL
CALLED
CALLING
CAPACITY
CASCADE
CASCADED
CASE
CAST
CATALOG
CHAR
CHARACTER
CHECK
CLASS
CLOB
CLOSE
CLUSTER
CLUSTERED
CLUSTERING
CLUSTERS
COALESCE
COLLATE
COLLATION
COLLECTION
COLUMN
```

```
COLUMNS
COMBINE
COMMENT
COMMIT
COMPACT
COMPILE
COMPRESS
CONDITION
CONFLICT
CONNECT
CONNECTION
CONSISTENCY
CONSISTENT
CONSTRAINT
CONSTRAINTS
CONSTRUCTOR
CONSUMED
CONTINUE
CONVERT
COPY
CORRESPONDING
COUNT
COUNTER
CREATE
CROSS
CUBE
CURRENT
CURSOR
CYCLE
DATA
DATABASE
DATE
DATETIME
DAY
DEALLOCATE
DEC
DECIMAL
DECLARE
DEFAULT
DEFERRABLE
DEFERRED
DEFINE
DEFINED
DEFINITION
```

```
DELETE
DELIMITED
DEPTH
DEREF
DESC
DESCRIBE
DESCRIPTOR
DETACH
DETERMINISTIC
DIAGNOSTICS
DIRECTORIES
DISABLE
DISCONNECT
DISTINCT
DISTRIBUTE
DO
DOMAIN
DOUBLE
DROP
DUMP
DURATION
DYNAMIC
EACH
ELEMENT
ELSE
ELSEIF
EMPTY
ENABLE
END
EQUAL
EQUALS
ERROR
ESCAPE
ESCAPED
EVAL
EVALUATE
EXCEEDED
EXCEPT
EXCEPTION
EXCEPTIONS
EXCLUSIVE
EXEC
EXECUTE
EXISTS
```

```
EXIT
EXPLAIN
EXPLODE
EXPORT
EXPRESSION
EXTENDED
EXTERNAL
EXTRACT
FAIL
FALSE
FAMILY
FETCH
FIELDS
FILE
FILTER
FILTERING
FINAL
FINISH
FIRST
FIXED
FLATTERN
FLOAT
FOR
FORCE
FOREIGN
FORMAT
FORWARD
FOUND
FREE
FROM
FULL
FUNCTION
FUNCTIONS
GENERAL
GENERATE
GET
GLOB
GLOBAL
GO
GOTO
GRANT
GREATER
GROUP
GROUPING
```

```
HANDLER
HASH
HAVE
HAVING
HEAP
HIDDEN
HOLD
HOUR
IDENTIFIED
IDENTITY
IF
IGNORE
IMMEDIATE
IMPORT
IN
INCLUDING
INCLUSIVE
INCREMENT
INCREMENTAL
INDEX
INDEXED
INDEXES
INDICATOR
INFINITE
INITIALLY
INLINE
INNER
INNTER
INOUT
INPUT
INSENSITIVE
INSERT
INSTEAD
INT
INTEGER
INTERSECT
INTERVAL
INTO
INVALIDATE
IS
ISOLATION
ITEM
ITEMS
ITERATE
```

```
JOIN
KEY
KEYS
LAG
LANGUAGE
LARGE
LAST
LATERAL
LEAD
LEADING
LEAVE
LEFT
LENGTH
LESS
LEVEL
LIKE
LIMIT
LIMITED
LINES
LIST
LOAD
LOCAL
LOCALTIME
LOCALTIMESTAMP
LOCATION
LOCATOR
LOCK
LOCKS
LOG
LOGED
LONG
LOOP
LOWER
MAP
MATCH
MATERIALIZED
MAX
MAXLEN
MEMBER
MERGE
METHOD
METRICS
MIN
MINUS
```

```
MINUTE
MISSING
MOD
MODE
MODIFIES
MODIFY
MODULE
MONTH
MULTI
MULTISET
NAME
NAMES
NATIONAL
NATURAL
NCHAR
NCLOB
NEW
NEXT
NO
NONE
NOT
NULL
NULLIF
NUMBER
NUMERIC
OBJECT
OF
OFFLINE
OFFSET
OLD
ON
ONLINE
ONLY
OPAQUE
OPEN
OPERATOR
OPTION
OR
ORDER
ORDINALITY
OTHER
OTHERS
OUT
OUTER
```

```
OUTPUT
OVER
OVERLAPS
OVERRIDE
OWNER
PAD
PARALLEL
PARAMETER
PARAMETERS
PARTIAL
PARTITION
PARTITIONED
PARTITIONS
PATH
PERCENT
PERCENTILE
PERMISSION
PERMISSIONS
PIPE
PIPELINED
PLAN
POOL
POSITION
PRECISION
PREPARE
PRESERVE
PRIMARY
PRIOR
PRIVATE
PRIVILEGES
PROCEDURE
PROCESSED
PROJECT
PROJECTION
PROPERTY
PROVISIONING
PUBLIC
PUT
QUERY
QUIT
QUORUM
RAISE
RANDOM
RANGE
```

```
RANK
RAW
READ
READS
REAL
REBUILD
RECORD
RECURSIVE
REDUCE
REF
REFERENCE
REFERENCES
REFERENCING
REGEXP
REGION
REINDEX
RELATIVE
RELEASE
REMAINDER
RENAME
REPEAT
REPLACE
REQUEST
RESET
RESIGNAL
RESOURCE
RESPONSE
RESTORE
RESTRICT
RESULT
RETURN
RETURNING
RETURNS
REVERSE
REVOKE
RIGHT
ROLE
ROLES
ROLLBACK
ROLLUP
ROUTINE
ROW
ROWS
RULE
```

```
RULES
SAMPLE
SATISFIES
SAVE
SAVEPOINT
SCAN
SCHEMA
SCOPE
SCROLL
SEARCH
SECOND
SECTION
SEGMENT
SEGMENTS
SELECT
SELF
SEMI
SENSITIVE
SEPARATE
SEQUENCE
SERIALIZABLE
SESSION
SET
SETS
SHARD
SHARE
SHARED
SHORT
SHOW
SIGNAL
SIMILAR
SIZE
SKEWED
SMALLINT
SNAPSHOT
SOME
SOURCE
SPACE
SPACES
SPARSE
SPECIFIC
SPECIFICTYPE
SPLIT
SQL
```

```
SQLCODE
SQLERROR
SQLEXCEPTION
SQLSTATE
SQLWARNING
START
STATE
STATIC
STATUS
STORAGE
STORE
STORED
STREAM
STRING
STRUCT
STYLE
SUB
SUBMULTISET
SUBPARTITION
SUBSTRING
SUBTYPE
SUM
SUPER
SYMMETRIC
SYNONYM
SYSTEM
TABLE
TABLESAMPLE
TEMP
TEMPORARY
TERMINATED
TEXT
THAN
THEN
THROUGHPUT
TIME
TIMESTAMP
TIMEZONE
TINYINT
TO
TOKEN
TOTAL
TOUCH
TRAILING
```

```
TRANSACTION
TRANSFORM
TRANSLATE
TRANSLATION
TREAT
TRIGGER
TRIM
TRUE
TRUNCATE
TTL
TUPLE
TYPE
UNDER
UNDO
UNION
UNIQUE
UNIT
UNKNOWN
UNLOGGED
UNNEST
UNPROCESSED
UNSIGNED
UNTIL
UPDATE
UPPER
URL
USAGE
USE
USER
USERS
USING
UUID
VACUUM
VALUE
VALUED
VALUES
VARCHAR
VARIABLE
VARIANCE
VARINT
VARYING
VIEW
VIEWS
VIRTUAL
```

```
VOID
WAIT
WHEN
WHENEVER
WHERE
WHILE
WINDOW
WITH
WITHIN
WITHOUT
WORK
WRAPPED
WRITE
YEAR
ZONE
```

AWS SDK for Java 1.x examples

This section contains example code for DAX applications using SDK for Java 1.x.

Topics

- Using DAX with AWS SDK for Java 1.x
- Modifying an existing SDK for Java 1.x application to use DAX
- Querying global secondary indexes with SDK for Java 1.x

Using DAX with AWS SDK for Java 1.x

Follow this procedure to run the Java sample for Amazon DynamoDB Accelerator (DAX) on your Amazon EC2 instance.

> **Note**
>
> These instructions are for applications using AWS SDK for Java 1.x. For applications using AWS SDK for Java 2.x, see Java and DAX.

To run the Java sample for DAX

1. Install the Java Development Kit (JDK).

```
sudo yum install -y java-devel
```

2. Download the AWS SDK for Java (`.zip` file), and then extract it.

```
wget http://sdk-for-java.amazonwebservices.com/latest/aws-java-sdk.zip

unzip aws-java-sdk.zip
```

3. Download the latest version of the DAX Java client (`.jar` file).

```
wget http://dax-sdk.s3-website-us-west-2.amazonaws.com/java/DaxJavaClient-
latest.jar
```

> **Note**
>
> The client for the DAX SDK for Java is available on Apache Maven. For more information, see Using the client as an Apache Maven dependency.

4. Set your CLASSPATH variable. In this example, replace *`sdkVersion`* with the actual version number of the AWS SDK for Java (for example, `1.11.112`).

```
export SDKVERSION=sdkVersion

export CLASSPATH=$(pwd)/TryDax/java:$(pwd)/DaxJavaClient-latest.jar:$(pwd)/
aws-java-sdk-$SDKVERSION/lib/aws-java-sdk-$SDKVERSION.jar:$(pwd)/aws-java-sdk-
$SDKVERSION/third-party/lib/*
```

5. Download the sample program source code (`.zip` file).

```
wget http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/samples/
TryDax.zip
```

When the download is complete, extract the source files.

```
unzip TryDax.zip
```

6. Navigate to the Java code directory and compile the code as follows.

```
cd TryDax/java/
```

```
javac TryDax*.java
```

7. Run the program.

```
java TryDax
```

You should see output similar to the following.

```
Creating a DynamoDB client

Attempting to create table; please wait...
Successfully created table.  Table status: ACTIVE
Writing data to the table...
Writing 10 items for partition key: 1
Writing 10 items for partition key: 2
Writing 10 items for partition key: 3
Writing 10 items for partition key: 4
Writing 10 items for partition key: 5
Writing 10 items for partition key: 6
Writing 10 items for partition key: 7
Writing 10 items for partition key: 8
Writing 10 items for partition key: 9
Writing 10 items for partition key: 10

Running GetItem, Scan, and Query tests...
First iteration of each test will result in cache misses
Next iterations are cache hits

GetItem test - partition key 1 and sort keys 1-10
 Total time: 136.681 ms - Avg time: 13.668 ms
 Total time: 122.632 ms - Avg time: 12.263 ms
 Total time: 167.762 ms - Avg time: 16.776 ms
 Total time: 108.130 ms - Avg time: 10.813 ms
 Total time: 137.890 ms - Avg time: 13.789 ms
Query test - partition key 5 and sort keys between 2 and 9
 Total time: 13.560 ms - Avg time: 2.712 ms
 Total time: 11.339 ms - Avg time: 2.268 ms
 Total time: 7.809 ms - Avg time: 1.562 ms
 Total time: 10.736 ms - Avg time: 2.147 ms
 Total time: 12.122 ms - Avg time: 2.424 ms
Scan test - all items in the table
 Total time: 58.952 ms - Avg time: 11.790 ms
 Total time: 25.507 ms - Avg time: 5.101 ms
```

```
 Total time: 37.660 ms - Avg time: 7.532 ms
 Total time: 26.781 ms - Avg time: 5.356 ms
 Total time: 46.076 ms - Avg time: 9.215 ms

Attempting to delete table; please wait...
Successfully deleted table.
```

Take note of the timing information—the number of milliseconds required for the `GetItem`, `Query`, and `Scan` tests.

8. In the previous step, you ran the program against the DynamoDB endpoint. Now run the program again, but this time, the `GetItem`, `Query`, and `Scan` operations are processed by your DAX cluster.

   To determine the endpoint for your DAX cluster, choose one of the following:

   - **Using the DynamoDB console** — Choose your DAX cluster. The cluster endpoint is shown on the console, as in the following example.

     ```
     dax://my-cluster.l6fzcv.dax-clusters.us-east-1.amazonaws.com
     ```

   - **Using the AWS CLI** — Enter the following command.

     ```
     aws dax describe-clusters --query "Clusters[*].ClusterDiscoveryEndpoint"
     ```

     The cluster endpoint is shown in the output, as in the following example.

     ```
     {
         "Address": "my-cluster.l6fzcv.dax-clusters.us-east-1.amazonaws.com",
         "Port": 8111,
         "URL": "dax://my-cluster.l6fzcv.dax-clusters.us-east-1.amazonaws.com"
     }
     ```

   Now run the program again, but this time, specify the cluster endpoint as a command line parameter.

   ```
   java TryDax dax://my-cluster.l6fzcv.dax-clusters.us-east-1.amazonaws.com
   ```

Look at the rest of the output, and take note of the timing information. The elapsed times for `GetItem`, `Query`, and `Scan` should be significantly lower with DAX than with DynamoDB.

For more information about this program, see the following sections:

- TryDax.java
- TryDaxHelper.java
- TryDaxTests.java

Using the client as an Apache Maven dependency

Follow these steps to use the client for the DAX SDK for Java in your application as a dependency.

To use the client as a Maven dependency

1. Download and install Apache Maven. For more information, see Downloading Apache Maven and Installing Apache Maven.
2. Add the client Maven dependency to your application's Project Object Model (POM) file. In this example, replace `x.x.x.x` with the actual version number of the client (for example, `1.0.200704.0`).

```
<!--Dependency:-->
<dependencies>
    <dependency>
     <groupId>com.amazonaws</groupId>
     <artifactId>amazon-dax-client</artifactId>
     <version>x.x.x.x</version>
    </dependency>
</dependencies>
```

TryDax.java

The `TryDax.java` file contains the `main` method. If you run the program with no command line parameters, it creates an Amazon DynamoDB client and uses that client for all API operations. If you specify a DynamoDB Accelerator (DAX) cluster endpoint on the command line, the program also creates a DAX client and uses it for `GetItem`, `Query`, and `Scan` operations.

You can modify the program in several ways:

- Use the DAX client instead of the DynamoDB client. For more information, see Java and DAX.
- Choose a different name for the test table.
- Modify the number of items written by changing the `helper.writeData` parameters. The second parameter is the number of partition keys, and the third parameter is the number of sort keys. By default, the program uses 1–10 for partition key values and 1–10 for sort key values, for a total of 100 items written to the table. For more information, see TryDaxHelper.java.
- Modify the number of `GetItem`, `Query`, and `Scan` tests, and modify their parameters.
- Comment out the lines containing `helper.createTable` and `helper.deleteTable` (if you don't want to create and delete the table each time you run the program).

> **Note**
>
> To run this program, you can set up Maven to use the client for the DAX SDK for Java and the AWS SDK for Java as dependencies. For more information, see Using the client as an Apache Maven dependency.
> Alternatively, you can download and include both the DAX Java client and the AWS SDK for Java in your classpath. See Java and DAX for an example of setting your `CLASSPATH` variable.

```
public class TryDax {

    public static void main(String[] args) throws Exception {

        TryDaxHelper helper = new TryDaxHelper();
        TryDaxTests tests = new TryDaxTests();

        DynamoDB ddbClient = helper.getDynamoDBClient();
        DynamoDB daxClient = null;
        if (args.length >= 1) {
            daxClient = helper.getDaxClient(args[0]);
        }

        String tableName = "TryDaxTable";
```

```
        System.out.println("Creating table...");
        helper.createTable(tableName, ddbClient);
        System.out.println("Populating table...");
        helper.writeData(tableName, ddbClient, 10, 10);

        DynamoDB testClient = null;
        if (daxClient != null) {
            testClient = daxClient;
        } else {
            testClient = ddbClient;
        }

        System.out.println("Running GetItem, Scan, and Query tests...");
        System.out.println("First iteration of each test will result in cache misses");
        System.out.println("Next iterations are cache hits\n");

        // GetItem
        tests.getItemTest(tableName, testClient, 1, 10, 5);

        // Query
        tests.queryTest(tableName, testClient, 5, 2, 9, 5);

        // Scan
        tests.scanTest(tableName, testClient, 5);

        helper.deleteTable(tableName, ddbClient);
    }

}
```

TryDaxHelper.java

The `TryDaxHelper.java` file contains utility methods.

The `getDynamoDBClient` and `getDaxClient` methods provide Amazon DynamoDB and DynamoDB Accelerator (DAX) clients. For control plane operations (`CreateTable`, `DeleteTable`) and write operations, the program uses the DynamoDB client. If you specify a DAX cluster endpoint, the main program creates a DAX client for performing read operations (`GetItem`, `Query`, `Scan`).

The other `TryDaxHelper` methods (`createTable`, `writeData`, `deleteTable`) are for setting up and tearing down the DynamoDB table and its data.

You can modify the program in several ways:

- Use different provisioned throughput settings for the table.
- Modify the size of each item written (see the `stringSize` variable in the `writeData` method).
- Modify the number of `GetItem`, `Query`, and `Scan` tests and their parameters.
- Comment out the lines containing `helper.CreateTable` and `helper.DeleteTable` (if you don't want to create and delete the table each time you run the program).

> **Note**
>
> To run this program, you can set up Maven to use the client for the DAX SDK for Java and the AWS SDK for Java as dependencies. For more information, see Using the client as an Apache Maven dependency.
> Or, you can download and include both the DAX Java client and the AWS SDK for Java in your classpath. See Java and DAX for an example of setting your `CLASSPATH` variable.

```
import com.amazon.dax.client.dynamodbv2.AmazonDaxClientBuilder;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;
import com.amazonaws.services.dynamodbv2.document.DynamoDB;
import com.amazonaws.services.dynamodbv2.document.Item;
import com.amazonaws.services.dynamodbv2.document.Table;
import com.amazonaws.services.dynamodbv2.model.AttributeDefinition;
import com.amazonaws.services.dynamodbv2.model.KeySchemaElement;
import com.amazonaws.services.dynamodbv2.model.KeyType;
import com.amazonaws.services.dynamodbv2.model.ProvisionedThroughput;
import com.amazonaws.services.dynamodbv2.model.ScalarAttributeType;
import com.amazonaws.util.EC2MetadataUtils;

public class TryDaxHelper {

    private static final String region = EC2MetadataUtils.getEC2InstanceRegion();

    DynamoDB getDynamoDBClient() {
        System.out.println("Creating a DynamoDB client");
        AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard()
                .withRegion(region)
```

```
                .build();
        return new DynamoDB(client);
    }

    DynamoDB getDaxClient(String daxEndpoint) {
        System.out.println("Creating a DAX client with cluster endpoint " +
daxEndpoint);
        AmazonDaxClientBuilder daxClientBuilder = AmazonDaxClientBuilder.standard();
        daxClientBuilder.withRegion(region).withEndpointConfiguration(daxEndpoint);
        AmazonDynamoDB client = daxClientBuilder.build();
        return new DynamoDB(client);
    }

    void createTable(String tableName, DynamoDB client) {
        Table table = client.getTable(tableName);
        try {
            System.out.println("Attempting to create table; please wait...");

            table = client.createTable(tableName,
                    Arrays.asList(
                            new KeySchemaElement("pk", KeyType.HASH), // Partition key
                            new KeySchemaElement("sk", KeyType.RANGE)), // Sort key
                    Arrays.asList(
                            new AttributeDefinition("pk", ScalarAttributeType.N),
                            new AttributeDefinition("sk", ScalarAttributeType.N)),
                    new ProvisionedThroughput(10L, 10L));
            table.waitForActive();
            System.out.println("Successfully created table.  Table status: " +
                    table.getDescription().getTableStatus());

        } catch (Exception e) {
            System.err.println("Unable to create table: ");
            e.printStackTrace();
        }
    }

    void writeData(String tableName, DynamoDB client, int pkmax, int skmax) {
        Table table = client.getTable(tableName);
        System.out.println("Writing data to the table...");

        int stringSize = 1000;
        StringBuilder sb = new StringBuilder(stringSize);
        for (int i = 0; i < stringSize; i++) {
            sb.append('X');
```

```
        }
        String someData = sb.toString();

        try {
            for (Integer ipk = 1; ipk <= pkmax; ipk++) {
                System.out.println(("Writing " + skmax + " items for partition key: " +
 ipk));
                for (Integer isk = 1; isk <= skmax; isk++) {
                    table.putItem(new Item()
                            .withPrimaryKey("pk", ipk, "sk", isk)
                            .withString("someData", someData));
                }
            }
        } catch (Exception e) {
            System.err.println("Unable to write item:");
            e.printStackTrace();
        }
    }

    void deleteTable(String tableName, DynamoDB client) {
        Table table = client.getTable(tableName);
        try {
            System.out.println("\nAttempting to delete table; please wait...");
            table.delete();
            table.waitForDelete();
            System.out.println("Successfully deleted table.");

        } catch (Exception e) {
            System.err.println("Unable to delete table: ");
            e.printStackTrace();
        }
    }

}
```

TryDaxTests.java

The `TryDaxTests.java` file contains methods that perform read operations against a test table in Amazon DynamoDB. These methods are not concerned with how they access the data (using either the DynamoDB client or the DAX client), so there is no need to modify the application logic.

You can modify the program in several ways:

- Modify the `queryTest` method so that it uses a different `KeyConditionExpression`.
- Add a `ScanFilter` to the `scanTest` method so that only some of the items are returned to you.

> **Note**
>
> To run this program, you can set up Maven to use the client for the DAX SDK for Java and the AWS SDK for Java as dependencies. For more information, see Using the client as an Apache Maven dependency.
> Or, you can download and include both the DAX Java client and the AWS SDK for Java in your classpath. See Java and DAX for an example of setting your `CLASSPATH` variable.

```
import java.util.Iterator;

import com.amazonaws.services.dynamodbv2.document.DynamoDB;
import com.amazonaws.services.dynamodbv2.document.Item;
import com.amazonaws.services.dynamodbv2.document.ItemCollection;
import com.amazonaws.services.dynamodbv2.document.QueryOutcome;
import com.amazonaws.services.dynamodbv2.document.ScanOutcome;
import com.amazonaws.services.dynamodbv2.document.Table;
import com.amazonaws.services.dynamodbv2.document.spec.QuerySpec;

public class TryDaxTests {

    void getItemTest(String tableName, DynamoDB client, int pk, int sk, int iterations)
 {
        long startTime, endTime;
        System.out.println("GetItem test - partition key " + pk + " and sort keys 1-" +
 sk);
        Table table = client.getTable(tableName);

        for (int i = 0; i < iterations; i++) {
            startTime = System.nanoTime();
            try {
                for (Integer ipk = 1; ipk <= pk; ipk++) {
                    for (Integer isk = 1; isk <= sk; isk++) {
                        table.getItem("pk", ipk, "sk", isk);
                    }
                }
```

```
            } catch (Exception e) {
                System.err.println("Unable to get item:");
                e.printStackTrace();
            }
            endTime = System.nanoTime();
            printTime(startTime, endTime, pk * sk);
        }
    }

    void queryTest(String tableName, DynamoDB client, int pk, int sk1, int sk2, int
iterations) {
        long startTime, endTime;
        System.out.println("Query test - partition key " + pk + " and sort keys between
" + sk1 + " and " + sk2);
        Table table = client.getTable(tableName);

        HashMap<String, Object> valueMap = new HashMap<String, Object>();
        valueMap.put(":pkval", pk);
        valueMap.put(":skval1", sk1);
        valueMap.put(":skval2", sk2);

        QuerySpec spec = new QuerySpec()
                .withKeyConditionExpression("pk = :pkval and sk between :skval1
and :skval2")
                .withValueMap(valueMap);

        for (int i = 0; i < iterations; i++) {
            startTime = System.nanoTime();
            ItemCollection<QueryOutcome> items = table.query(spec);

            try {
                Iterator<Item> iter = items.iterator();
                while (iter.hasNext()) {
                    iter.next();
                }
            } catch (Exception e) {
                System.err.println("Unable to query table:");
                e.printStackTrace();
            }
            endTime = System.nanoTime();
            printTime(startTime, endTime, iterations);
        }
    }
```

```
    void scanTest(String tableName, DynamoDB client, int iterations) {
        long startTime, endTime;
        System.out.println("Scan test - all items in the table");
        Table table = client.getTable(tableName);

        for (int i = 0; i < iterations; i++) {
            startTime = System.nanoTime();
            ItemCollection<ScanOutcome> items = table.scan();
            try {

                Iterator<Item> iter = items.iterator();
                while (iter.hasNext()) {
                    iter.next();
                }
            } catch (Exception e) {
                System.err.println("Unable to scan table:");
                e.printStackTrace();
            }
            endTime = System.nanoTime();
            printTime(startTime, endTime, iterations);
        }
    }

    public void printTime(long startTime, long endTime, int iterations) {
        System.out.format("\tTotal time: %.3f ms - ", (endTime - startTime) /
 (1000000.0));
        System.out.format("Avg time: %.3f ms\n", (endTime - startTime) / (iterations *
 1000000.0));
    }
}
```

Modifying an existing SDK for Java 1.x application to use DAX

If you already have a Java application that uses Amazon DynamoDB, you have to modify it so that it can access your DynamoDB Accelerator (DAX) cluster. You don't have to rewrite the entire application because the DAX Java client is similar to the DynamoDB low-level client included in the AWS SDK for Java.

> **Note**
>
> These instructions are for applications using AWS SDK for Java 1.x. For applications using AWS SDK for Java 2.x, see Modifying an existing application to use DAX.

Suppose that you have a DynamoDB table named `Music`. The partition key for the table is `Artist`, and its sort key is `SongTitle`. The following program reads an item directly from the `Music` table.

```
import java.util.HashMap;

import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;
import com.amazonaws.services.dynamodbv2.model.AttributeValue;
import com.amazonaws.services.dynamodbv2.model.GetItemRequest;
import com.amazonaws.services.dynamodbv2.model.GetItemResult;

public class GetMusicItem {

    public static void main(String[] args) throws Exception {

        // Create a DynamoDB client
        AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();

        HashMap<String, AttributeValue> key = new HashMap<String, AttributeValue>();
        key.put("Artist", new AttributeValue().withS("No One You Know"));
        key.put("SongTitle", new AttributeValue().withS("Scared of My Shadow"));

        GetItemRequest request = new GetItemRequest()
            .withTableName("Music").withKey(key);

        try {
            System.out.println("Attempting to read the item...");
            GetItemResult result = client.getItem(request);
            System.out.println("GetItem succeeded: " + result);

        } catch (Exception e) {
            System.err.println("Unable to read item");
            System.err.println(e.getMessage());
        }
    }
}
```

To modify the program, replace the DynamoDB client with a DAX client.

```
import java.util.HashMap;

import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazon.dax.client.dynamodbv2.AmazonDaxClientBuilder;
import com.amazonaws.services.dynamodbv2.model.AttributeValue;
import com.amazonaws.services.dynamodbv2.model.GetItemRequest;
import com.amazonaws.services.dynamodbv2.model.GetItemResult;

public class GetMusicItem {

    public static void main(String[] args) throws Exception {

    //Create a DAX client

    AmazonDaxClientBuilder daxClientBuilder = AmazonDaxClientBuilder.standard();
    daxClientBuilder.withRegion("us-
east-1").withEndpointConfiguration("mydaxcluster.2cmrwl.clustercfg.dax.use1.cache.amazonaws.com
    AmazonDynamoDB client = daxClientBuilder.build();


       /*
       ** ...
       ** Remaining code omitted (it is identical)
       ** ...
       */

    }
}
```

Using the DynamoDB document API

The AWS SDK for Java provides a document interface for DynamoDB. The document API acts as a wrapper around the low-level DynamoDB client. For more information, see Document interfaces.

The document interface can also be used with the low-level DAX client, as shown in the following example.

```
import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazon.dax.client.dynamodbv2.AmazonDaxClientBuilder;
import com.amazonaws.services.dynamodbv2.document.DynamoDB;
import com.amazonaws.services.dynamodbv2.document.GetItemOutcome;
```

```
import com.amazonaws.services.dynamodbv2.document.Table;

public class GetMusicItemWithDocumentApi {

    public static void main(String[] args) throws Exception {

        //Create a DAX client

        AmazonDaxClientBuilder daxClientBuilder = AmazonDaxClientBuilder.standard();
        daxClientBuilder.withRegion("us-
east-1").withEndpointConfiguration("mydaxcluster.2cmrwl.clustercfg.dax.use1.cache.amazonaws.com
        AmazonDynamoDB client = daxClientBuilder.build();

        // Document client wrapper
        DynamoDB docClient = new DynamoDB(client);

        Table table = docClient.getTable("Music");

        try {
            System.out.println("Attempting to read the item...");
            GetItemOutcome outcome = table.tgetItemOutcome(
                "Artist", "No One You Know",
                "SongTitle", "Scared of My Shadow");
            System.out.println(outcome.getItem());
            System.out.println("GetItem succeeded: " + outcome);
        } catch (Exception e) {
            System.err.println("Unable to read item");
            System.err.println(e.getMessage());
        }

    }
}
```

DAX async client

The `AmazonDaxClient` is synchronous. For a long-running DAX API operation, such as a `Scan` of a large table, this can block program execution until the operation is complete. If your program needs to perform other work while a DAX API operation is in progress, you can use `ClusterDaxAsyncClient` instead.

The following program shows how to use `ClusterDaxAsyncClient`, along with Java `Future`, to implement a non-blocking solution.

```
import java.util.concurrent.ExecutionException;
import java.util.concurrent.Future;

import com.amazon.dax.client.dynamodbv2.ClientConfig;
import com.amazon.dax.client.dynamodbv2.ClusterDaxAsyncClient;
import com.amazonaws.auth.profile.ProfileCredentialsProvider;
import com.amazonaws.handlers.AsyncHandler;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBAsync;
import com.amazonaws.services.dynamodbv2.model.AttributeValue;
import com.amazonaws.services.dynamodbv2.model.GetItemRequest;
import com.amazonaws.services.dynamodbv2.model.GetItemResult;

public class DaxAsyncClientDemo {
 public static void main(String[] args) throws Exception {

  ClientConfig daxConfig = new ClientConfig().withCredentialsProvider(new
 ProfileCredentialsProvider())
    .withEndpoints("mydaxcluster.2cmrwl.clustercfg.dax.use1.cache.amazonaws.com:8111");

  AmazonDynamoDBAsync client = new ClusterDaxAsyncClient(daxConfig);

  HashMap<String, AttributeValue> key = new HashMap<String, AttributeValue>();
  key.put("Artist", new AttributeValue().withS("No One You Know"));
  key.put("SongTitle", new AttributeValue().withS("Scared of My Shadow"));

  GetItemRequest request = new GetItemRequest()
    .withTableName("Music").withKey(key);

  // Java Futures
  Future<GetItemResult> call = client.getItemAsync(request);
  while (!call.isDone()) {
   // Do other processing while you're waiting for the response
   System.out.println("Doing something else for a few seconds...");
   Thread.sleep(3000);
  }
  // The results should be ready by now

  try {
   call.get();

  } catch (ExecutionException ee) {
   // Futures always wrap errors as an ExecutionException.
   // The *real* exception is stored as the cause of the
```

```
   // ExecutionException
   Throwable exception = ee.getCause();
   System.out.println("Error getting item: " + exception.getMessage());
  }

  // Async callbacks
  call = client.getItemAsync(request, new AsyncHandler<GetItemRequest, GetItemResult>()
 {

   @Override
   public void onSuccess(GetItemRequest request, GetItemResult getItemResult) {
    System.out.println("Result: " + getItemResult);
   }

   @Override
   public void onError(Exception e) {
    System.out.println("Unable to read item");
    System.err.println(e.getMessage());
    // Callers can also test if exception is an instance of
    // AmazonServiceException or AmazonClientException and cast
    // it to get additional information
   }

  });
  call.get();

 }
}
```

Querying global secondary indexes with SDK for Java 1.x

You can use Amazon DynamoDB Accelerator (DAX) to query global secondary indexes using DynamoDB programmatic interfaces.

The following example demonstrates how to use DAX to query the `CreateDateIndex` global secondary index that is created in Example: Global secondary indexes using the AWS SDK for Java document API.

The `DAXClient` class instantiates the client objects that are needed to interact with the DynamoDB programming interfaces.

```
import com.amazon.dax.client.dynamodbv2.AmazonDaxClientBuilder;
```

```
import software.amazon.dynamodb.datamodeling.DynamoDBMapper;
import software.amazon.dynamodb.document.DynamoDB;
import com.amazonaws.util.EC2MetadataUtils;
import software.amazon.dynamodb.AmazonDynamoDB;

public class DaxClient {

 private static final String region = EC2MetadataUtils.getEC2InstanceRegion();

 DynamoDB getDaxDocClient(String daxEndpoint) {
  System.out.println("Creating a DAX client with cluster endpoint " + daxEndpoint);
  AmazonDaxClientBuilder daxClientBuilder = AmazonDaxClientBuilder.standard();

  daxClientBuilder.withRegion(region).withEndpointConfiguration(daxEndpoint);
  AmazonDynamoDB client = daxClientBuilder.build();

  return new DynamoDB(client);
 }

 DynamoDBMapper getDaxMapperClient(String daxEndpoint) {
  System.out.println("Creating a DAX client with cluster endpoint " + daxEndpoint);
  AmazonDaxClientBuilder daxClientBuilder = AmazonDaxClientBuilder.standard();

  daxClientBuilder.withRegion(region).withEndpointConfiguration(daxEndpoint);
  AmazonDynamoDB client = daxClientBuilder.build();

  return new DynamoDBMapper(client);
 }
}
```

You can query a global secondary index in the following ways:

- Use the `queryIndex` method on the `QueryIndexDax` class defined in the following example. The `QueryIndexDax` takes as a parameter the client object that is returned by the `getDaxDocClient` method on the `DaxClient` class.
- If you are using the object persistence interface, use the `queryIndexMapper` method on the `QueryIndexDax` class defined in the following example. The `queryIndexMapper` takes as a parameter the client object that is returned by the `getDaxMapperClient` method defined on the `DaxClient` class.

```
import java.util.Iterator;
```

```
import software.amazon.dynamodb.datamodeling.DynamoDBMapper;
import java.util.List;
import software.amazon.dynamodb.datamodeling.DynamoDBQueryExpression;
import software.amazon.dynamodb.model.AttributeValue;
import java.util.HashMap;
import software.amazon.dynamodb.document.Item;
import software.amazon.dynamodb.document.utils.ValueMap;
import software.amazon.dynamodb.document.spec.QuerySpec;
import software.amazon.dynamodb.document.QueryOutcome;
import software.amazon.dynamodb.document.ItemCollection;
import software.amazon.dynamodb.document.Index;
import software.amazon.dynamodb.document.Table;
import software.amazon.dynamodb.document.DynamoDB;

public class QueryIndexDax {

 //This is used to query Index using the low-level interface.
 public static void queryIndex(DynamoDB client, String tableName, String indexName) {
  Table table = client.getTable(tableName);

  System.out.println("\n*********************************************************
\n");
  System.out.print("Querying index " + indexName + "...");

  Index index = table.getIndex(indexName);

  ItemCollection<QueryOutcome> items = null;

  QuerySpec querySpec = new QuerySpec();

  if (indexName == "CreateDateIndex") {
   System.out.println("Issues filed on 2013-11-01");
   querySpec.withKeyConditionExpression("CreateDate = :v_date and
 begins_with(IssueId, :v_issue)")
     .withValueMap(new ValueMap().withString(":v_date",
 "2013-11-01").withString(":v_issue", "A-"));
   items = index.query(querySpec);
  } else {
   System.out.println("\nNo valid index name provided");
   return;
  }

  Iterator<Item> iterator = items.iterator();
```

```
  System.out.println("Query: printing results...");

  while (iterator.hasNext()) {
   System.out.println(iterator.next().toJSONPretty());
  }

 }

 //This is used to query Index using the high-level mapper interface.
 public static void queryIndexMapper(DynamoDBMapper mapper, String tableName, String
 indexName) {
  HashMap<String, AttributeValue> eav = new HashMap<String, AttributeValue>();
  eav.put(":v_date", new AttributeValue().withS("2013-11-01"));
  eav.put(":v_issue", new AttributeValue().withS("A-"));
  DynamoDBQueryExpression<CreateDate> queryExpression = new
 DynamoDBQueryExpression<CreateDate>()
     .withIndexName("CreateDateIndex").withConsistentRead(false)
     .withKeyConditionExpression("CreateDate = :v_date and
 begins_with(IssueId, :v_issue)")
     .withExpressionAttributeValues(eav);

  List<CreateDate> items = mapper.query(CreateDate.class, queryExpression);
  Iterator<CreateDate> iterator = items.iterator();

  System.out.println("Query: printing results...");

  while (iterator.hasNext()) {
   CreateDate iterObj = iterator.next();
   System.out.println(iterObj.getCreateDate());
   System.out.println(iterObj.getIssueId());
  }
 }
}
```

The class definition below represents the Issues table and is used in the `queryIndexMapper` method.

```
import software.amazon.dynamodb.datamodeling.DynamoDBTable;
import software.amazon.dynamodb.datamodeling.DynamoDBIndexHashKey;
import software.amazon.dynamodb.datamodeling.DynamoDBIndexRangeKey;
import software.amazon.dynamodb.datamodeling.DynamoDBHashKey;

@DynamoDBTable(tableName = "Issues")
```

```
public class CreateDate {
 private String createDate;
 @DynamoDBHashKey(attributeName = "IssueId")
 private String issueId;

 @DynamoDBIndexHashKey(globalSecondaryIndexName = "CreateDateIndex", attributeName =
 "CreateDate")
 public String getCreateDate() {
  return createDate;
 }

 public void setCreateDate(String createDate) {
  this.createDate = createDate;
 }

 @DynamoDBIndexRangeKey(globalSecondaryIndexName = "CreateDateIndex", attributeName =
 "IssueId")
 public String getIssueId() {
  return issueId;
 }

 public void setIssueId(String issueId) {
  this.issueId = issueId;
 }
}
```

AWS SDK for Go 1.x examples

This section contains example code for DAX applications using Go 1.x.

Topics

- DAX SDK for Go

DAX SDK for Go

Follow this procedure to run the Amazon DynamoDB Accelerator (DAX) SDK for Go sample application on your Amazon EC2 instance.

To run the SDK for Go sample for DAX

1. Set up the SDK for Go on your Amazon EC2 instance:

a. Install the Go programming language (`Golang`).

```
sudo yum install -y golang
```

b. Test that Golang is installed and running correctly.

```
go version
```

A message like this should appear.

```
go version go1.15.5 linux/amd64
```

The remaining instructions rely on module support, which became the default with Go version 1.13.

2. Install the sample Golang application.

```
go get github.com/aws-samples/aws-dax-go-sample
```

3. Run the following Golang programs. The first program creates a DynamoDB table named `TryDaxGoTable`. The second program writes data to the table.

```
go run ~/go/pkg/mod/github.com/aws-samples/aws-dax-go-sample@v1.0.2/try_dax.go -
service dynamodb -command create-table
```

```
go run ~/go/pkg/mod/github.com/aws-samples/aws-dax-go-sample@v1.0.2/try_dax.go -
service dynamodb -command put-item
```

4. Run the following Golang programs.

```
go run ~/go/pkg/mod/github.com/aws-samples/aws-dax-go-sample@v1.0.2/try_dax.go -
service dynamodb -command get-item
```

```
go run ~/go/pkg/mod/github.com/aws-samples/aws-dax-go-sample@v1.0.2/try_dax.go -
service dynamodb -command query
```

```
go run ~/go/pkg/mod/github.com/aws-samples/aws-dax-go-sample@v1.0.2/try_dax.go -
service dynamodb -command scan
```

Take note of the timing information—the number of milliseconds required for the `GetItem`, `Query`, and `Scan` tests.

5. In the previous step, you ran the programs against the DynamoDB endpoint. Now, run the programs again, but this time, the `GetItem`, `Query`, and `Scan` operations are processed by your DAX cluster.

   To determine the endpoint for your DAX cluster, choose one of the following:

   - **Using the DynamoDB console** — Choose your DAX cluster. The cluster endpoint is shown on the console, as in the following example.

     ```
     dax://my-cluster.l6fzcv.dax-clusters.us-east-1.amazonaws.com
     ```

   - **Using the AWS CLI** — Enter the following command.

     ```
     aws dax describe-clusters --query "Clusters[*].ClusterDiscoveryEndpoint"
     ```

     The cluster endpoint is shown in the output, as in the following example.

     ```
     {
         "Address": "my-cluster.l6fzcv.dax-clusters.us-east-1.amazonaws.com",
         "Port": 8111,
         "URL": "dax://my-cluster.l6fzcv.dax-clusters.us-east-1.amazonaws.com"
     }
     ```

   Now run the programs again, but this time, specify the cluster endpoint as a command line parameter.

   ```
   go run ~/go/pkg/mod/github.com/aws-samples/aws-dax-go-sample@v1.0.2/try_dax.go
    -service dax -command get-item -endpoint my-cluster.l6fzcv.dax-clusters.us-
   east-1.amazonaws.com:8111
   ```

   ```
   go run ~/go/pkg/mod/github.com/aws-samples/aws-dax-go-sample@v1.0.2/try_dax.go
    -service dax -command query -endpoint my-cluster.l6fzcv.dax-clusters.us-
   east-1.amazonaws.com:8111
   ```

```
go run ~/go/pkg/mod/github.com/aws-samples/aws-dax-go-sample@v1.0.2/try_dax.go
 -service dax -command scan -endpoint my-cluster.l6fzcv.dax-clusters.us-
east-1.amazonaws.com:8111
```

Look at the rest of the output, and take note of the timing information. The elapsed times for `GetItem`, `Query`, and `Scan` should be significantly lower with DAX than with DynamoDB.

6. Run the following Golang program to delete `TryDaxGoTable`.

```
go run ~/go/pkg/mod/github.com/aws-samples/aws-dax-go-sample@v1.0.2/try_dax.go -
service dynamodb -command delete-table
```

AWS SDK for Node.js 2.x examples

This section contains example code for DAX applications using AWS SDK for Node.js 2.x.

Topics

- Node.js and DAX

Node.js and DAX

Follow these steps to run the Node.js sample application on your Amazon EC2 instance.

To run the Node.js sample for DAX

1. Set up Node.js on your Amazon EC2 instance, as follows:

   a. Install node version manager (nvm).

   ```
   curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.35.3/install.sh | bash
   ```

   b. Use nvm to install Node.js.

   ```
   nvm install 12.16.3
   ```

   c. Test that Node.js is installed and running correctly.

   ```
   node -e "console.log('Running Node.js ' + process.version)"
   ```

This should display the following message.

```
Running Node.js v12.16.3
```

2. Install the DAX Node.js client using the node package manager (`npm`).

```
npm install amazon-dax-client
```

3. Download the sample program source code (`.zip` file).

```
wget http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/samples/
TryDax.zip
```

When the download is complete, extract the source files.

```
unzip TryDax.zip
```

4. Run the following Node.js programs. The first program creates an Amazon DynamoDB table named `TryDaxTable`. The second program writes data to the table.

```
node 01-create-table.js
node 02-write-data.js
```

5. Run the following Node.js programs.

```
node 03-getitem-test.js
node 04-query-test.js
node 05-scan-test.js
```

Take note of the timing information—the number of milliseconds required for the `GetItem`, `Query`, and `Scan` tests.

6. In the previous step, you ran the programs against the DynamoDB endpoint. Run the programs again, but this time, the `GetItem`, `Query` and `Scan` operations are processed by your DAX cluster.

To determine the endpoint for your DAX cluster, choose one of the following.

- **Using the DynamoDB console**—Choose your DAX cluster. The cluster endpoint is shown on the console, as in the following example.

```
dax://my-cluster.l6fzcv.dax-clusters.us-east-1.amazonaws.com
```

- **Using the AWS CLI**—Enter the following command.

```
aws dax describe-clusters --query "Clusters[*].ClusterDiscoveryEndpoint"
```

The cluster endpoint is shown in the output, as in the following example.

```
{
    "Address": "my-cluster.l6fzcv.dax-clusters.us-east-1.amazonaws.com",
    "Port": 8111,
    "URL": "dax://my-cluster.l6fzcv.dax-clusters.us-east-1.amazonaws.com"
}
```

Now run the programs again, but this time, specify the cluster endpoint as a command line parameter.

```
node 03-getitem-test.js dax://my-cluster.l6fzcv.dax-clusters.us-
east-1.amazonaws.com
node 04-query-test.js dax://my-cluster.l6fzcv.dax-clusters.us-east-1.amazonaws.com
node 05-scan-test.js dax://my-cluster.l6fzcv.dax-clusters.us-east-1.amazonaws.com
```

Look at the rest of the output, and take note of the timing information. The elapsed times for `GetItem`, `Query`, and `Scan` should be significantly lower with DAX than with DynamoDB.

7. Run the following Node.js program to delete `TryDaxTable`.

```
node 06-delete-table
```

For more information about these programs, see the following sections:

- 01-create-table.js
- 02-write-data.js
- 03-getitem-test.js
- 04-query-test.js
- 05-scan-test.js

- 06-delete-table.js

01-create-table.js

The `01-create-table.js` program creates a table (`TryDaxTable`). The remaining Node.js programs in this section depend on this table.

```
const AmazonDaxClient = require("amazon-dax-client");
var AWS = require("aws-sdk");

var region = "us-west-2";

AWS.config.update({
  region: region,
});

var dynamodb = new AWS.DynamoDB(); //low-level client

var tableName = "TryDaxTable";

var params = {
  TableName: tableName,
  KeySchema: [
    { AttributeName: "pk", KeyType: "HASH" }, //Partition key
    { AttributeName: "sk", KeyType: "RANGE" }, //Sort key
  ],
  AttributeDefinitions: [
    { AttributeName: "pk", AttributeType: "N" },
    { AttributeName: "sk", AttributeType: "N" },
  ],
  ProvisionedThroughput: {
    ReadCapacityUnits: 10,
    WriteCapacityUnits: 10,
  },
};

dynamodb.createTable(params, function (err, data) {
  if (err) {
    console.error(
      "Unable to create table. Error JSON:",
      JSON.stringify(err, null, 2)
    );
  } else {
```

```
    console.log(
      "Created table. Table description JSON:",
      JSON.stringify(data, null, 2)
    );
  }
});
```

02-write-data.js

The `02-write-data.js` program writes test data to `TryDaxTable`.

```
const AmazonDaxClient = require("amazon-dax-client");
var AWS = require("aws-sdk");

var region = "us-west-2";

AWS.config.update({
  region: region,
});

var ddbClient = new AWS.DynamoDB.DocumentClient();

var tableName = "TryDaxTable";

var someData = "X".repeat(1000);
var pkmax = 10;
var skmax = 10;

for (var ipk = 1; ipk <= pkmax; ipk++) {
  for (var isk = 1; isk <= skmax; isk++) {
    var params = {
      TableName: tableName,
      Item: {
        pk: ipk,
        sk: isk,
        someData: someData,
      },
    };

    //
    //put item
```

```
    ddbClient.put(params, function (err, data) {
      if (err) {
        console.error("Unable to write data: ", JSON.stringify(err, null, 2));
      } else {
        console.log("PutItem succeeded");
      }
    });
  }
}
```

03-getitem-test.js

The `03-getitem-test.js` program performs `GetItem` operations on `TryDaxTable`.

```
const AmazonDaxClient = require("amazon-dax-client");
var AWS = require("aws-sdk");

var region = "us-west-2";

AWS.config.update({
  region: region,
});

var ddbClient = new AWS.DynamoDB.DocumentClient();
var daxClient = null;

if (process.argv.length > 2) {
  var dax = new AmazonDaxClient({
    endpoints: [process.argv[2]],
    region: region,
  });
  daxClient = new AWS.DynamoDB.DocumentClient({ service: dax });
}

var client = daxClient != null ? daxClient : ddbClient;
var tableName = "TryDaxTable";

var pk = 1;
var sk = 10;
var iterations = 5;

for (var i = 0; i < iterations; i++) {
```

```
  var startTime = new Date().getTime();

  for (var ipk = 1; ipk <= pk; ipk++) {
    for (var isk = 1; isk <= sk; isk++) {
      var params = {
        TableName: tableName,
        Key: {
          pk: ipk,
          sk: isk,
        },
      };

      client.get(params, function (err, data) {
        if (err) {
          console.error(
            "Unable to read item. Error JSON:",
            JSON.stringify(err, null, 2)
          );
        } else {
          // GetItem succeeded
        }
      });
    }
  }

  var endTime = new Date().getTime();
  console.log(
    "\tTotal time: ",
    endTime - startTime,
    "ms - Avg time: ",
    (endTime - startTime) / iterations,
    "ms"
  );
}
```

04-query-test.js

The `04-query-test.js` program performs `Query` operations on `TryDaxTable`.

```
const AmazonDaxClient = require("amazon-dax-client");
var AWS = require("aws-sdk");
```

```
var region = "us-west-2";

AWS.config.update({
  region: region,
});

var ddbClient = new AWS.DynamoDB.DocumentClient();
var daxClient = null;

if (process.argv.length > 2) {
  var dax = new AmazonDaxClient({
    endpoints: [process.argv[2]],
    region: region,
  });
  daxClient = new AWS.DynamoDB.DocumentClient({ service: dax });
}

var client = daxClient != null ? daxClient : ddbClient;
var tableName = "TryDaxTable";

var pk = 5;
var sk1 = 2;
var sk2 = 9;
var iterations = 5;

var params = {
  TableName: tableName,
  KeyConditionExpression: "pk = :pkval and sk between :skval1 and :skval2",
  ExpressionAttributeValues: {
    ":pkval": pk,
    ":skval1": sk1,
    ":skval2": sk2,
  },
};

for (var i = 0; i < iterations; i++) {
  var startTime = new Date().getTime();

  client.query(params, function (err, data) {
    if (err) {
      console.error(
        "Unable to read item. Error JSON:",
        JSON.stringify(err, null, 2)
      );
```

```
    } else {
      // Query succeeded
    }
  });

  var endTime = new Date().getTime();
  console.log(
    "\tTotal time: ",
    endTime - startTime,
    "ms - Avg time: ",
    (endTime - startTime) / iterations,
    "ms"
  );
}
```

05-scan-test.js

The `05-scan-test.js` program performs `Scan` operations on `TryDaxTable`.

```
const AmazonDaxClient = require("amazon-dax-client");
var AWS = require("aws-sdk");

var region = "us-west-2";

AWS.config.update({
  region: region,
});

var ddbClient = new AWS.DynamoDB.DocumentClient();
var daxClient = null;

if (process.argv.length > 2) {
  var dax = new AmazonDaxClient({
    endpoints: [process.argv[2]],
    region: region,
  });
  daxClient = new AWS.DynamoDB.DocumentClient({ service: dax });
}

var client = daxClient != null ? daxClient : ddbClient;
var tableName = "TryDaxTable";
```

```
var iterations = 5;

var params = {
  TableName: tableName,
};
var startTime = new Date().getTime();
for (var i = 0; i < iterations; i++) {
  client.scan(params, function (err, data) {
    if (err) {
      console.error(
        "Unable to read item. Error JSON:",
        JSON.stringify(err, null, 2)
      );
    } else {
      // Scan succeeded
    }
  });
}

var endTime = new Date().getTime();
console.log(
  "\tTotal time: ",
  endTime - startTime,
  "ms - Avg time: ",
  (endTime - startTime) / iterations,
  "ms"
);
```

06-delete-table.js

The `06-delete-table.js` program deletes `TryDaxTable`. Run this program after you have finished testing.

```
const AmazonDaxClient = require("amazon-dax-client");
var AWS = require("aws-sdk");

var region = "us-west-2";

AWS.config.update({
  region: region,
});
```

```
var dynamodb = new AWS.DynamoDB(); //low-level client

var tableName = "TryDaxTable";

var params = {
  TableName: tableName,
};

dynamodb.deleteTable(params, function (err, data) {
  if (err) {
    console.error(
      "Unable to delete table. Error JSON:",
      JSON.stringify(err, null, 2)
    );
  } else {
    console.log(
      "Deleted table. Table description JSON:",
      JSON.stringify(data, null, 2)
    );
  }
});
```

Document history for DynamoDB

The following table describes the important changes in each release of the *DynamoDB Developer Guide* from July 3, 2018 onward. For notification about updates to this documentation, you can subscribe to the RSS feed (at the top left corner of this page).

Change	Description	Date
DAX now supports AWS PrivateLink in commercial and China Regions	Added guidance for using AWS PrivateLink with DynamoDB Accelerator (DAX) to help users establish private, secure connectivity to DAX control plane endpoints within the same AWS Region. For more information, see Generate infrastructure code for Amazon DynamoDB using Console-to-Code.	October 30, 2025
DynamoDB now supports Internet Protocol version 6 (IPv6)	DynamoDB now supports Internet Protocol version 6 (IPv6) addresses in your Amazon VPC when you connect to DynamoDB tables, streams, and DAX, including with AWS PrivateLink Gateway and Interface endpoints. You can use IPv6 to simplify your network stack and meet compliance requirements with a network that supports both IPv4 and IPv6.	October 3, 2025

Added documentation for the new Console-to-Code feature for DynamoDB	Console-to-Code transforms manual DynamoDB table creation steps in the AWS Management Console into automated, reproducible infrastructure code across multiple formats like AWS CDK and CloudFormation. For more information, see Generate infrastructure code for Amazon DynamoDB using Console-to-Code.	September 23, 2025
Released DynamoDB Local version 3.1.0	This update improves performance for PartiQL queries, including Joda-time dependency.	September 14, 2025

DynamoDB has enhanced its throttling diagnostics capabilities with a new CloudWatch Contributor Insights mode, added throttling exception details, and targeted CloudWatch metrics	DynamoDB provides enhanced throttling diagnostics with a new `Throttled keys` only mode in CloudWatch Contributor Insights that specifically tracks the most throttled partition keys, more detailed throttling exceptions that include structured `ThrottlingReason` fields, providing specific reason codes and resource ARNs to help identify exactly what resource is throttled and why, and 16 new targeted CloudWatch metrics that correspond to different throttling scenario, for precise monitoring of specific throttling events. For more information, see Troubleshooting throttling in DynamoDB.	August 14, 2025

Increased limit for switching between capacity modes	Increased the limit for switching a table from provisioned to on-demand capacity mode from once every 24 hours to four times every 24 hours. This change provides more flexibility for workloads that require loading large volumes of data multiple times per day or need to accommodate unexpected traffic increases . For more information, see Considerations when switching capacity modes in DynamoDB.	August 12, 2025
Released DynamoDB Local version 3.0, which migrates from AWS SDK for Java V1 to V2	This update removes the V1 dependency, improves performance, and aligns with the AWS SDK V2 package structure.	July 17, 2025
Added support for optimized shard discovery in DynamoDB Streams	The new shard discovery mechanism reduces initial discovery time, improves steady-state processin g performance, provides visibility into processin g status, and maintains consistent p100 latency across large-scale operation s. For more information, see AWS PrivateLink for Amazon DynamoDB Streams.	July 17, 2025

Added Multi-Region Strong Consistency (MRSC) support for DynamoDB global tables	DynamoDB now supports Multi-Region Strong Consistency (MRSC) for global tables with optional witness Region. MRSC allows strongly consistent reads across Regions with a Recovery Point Objective (RPO) of zero during regional failures. MRSC global tables must be deployed in exactly three Regions, with either three replicas or two replicas and one witness. For more information and available Regions, see Multi-Region strong consistency (MRSC).	June 30, 2025
Amazon DynamoDB Streams Kinesis Client Library (KCL) 2.x adapter	DynamoDB now supports an upgraded DynamoDB Streams adapter compatible with KCL 2.x and AWS SDK v2. For more information, see Migrating from KCL1.x to KCL 3.x.	June 11, 2025
Added support for running DynamoDB local in AWS CloudShell	You can now run DynamoDB local directly in your browser through the AWS Managemen t Console without downloadi ng or installing software. For more information, see Deploying DynamoDB locally on your computer.	May 19, 2025

Added support for r7i instance types in DAX	Expanded instance type options for DAX to provide more flexibility in performance and capacity. For more details about the new r7i instance types and their capabilities, see DAX encryption at rest.	April 29, 2025
New scoped-down full access policy `AmazonDynamoDBFullAccess_v2` replaces `AmazonDynamoDBFullAccess` .	Effective **April 25, 2025**, DynamoDB has deprecated the `AmazonDynamoDBFullAccess` policy and replaced it with a scoped-down policy named `AmazonDynamoDBFullAccess_v2` . For more information, see AWS managed policy: AmazonDynamoDBFullAccess_v2.	April 28, 2025

DynamoDB introduces AWS PrivateLink support for DynamoDB Streams	AWS added AWS PrivateLink support for DynamoDB Streams, enabling customers to invoke DynamoDB Streams APIs from within Amazon Virtual Private Cloud (VPC) using private network connectivity. This feature allows simplified private network access without traversing the public internet, supporting compliance and security requirements for DynamoDB workloads. For more information, see AWS PrivateLink for Amazon DynamoDB Streams.	March 26, 2025
DAX SDK For JS 3.x and Go 2.x is now available	DynamoDB Accelerator (DAX) SDK for JS 3.x is now available and is compatible with the AWS SDK for Java 3.x.	March 17, 2025
NoSQL Workbench 3.13.5 on-demand capacity mode for default table settings release	When you create a table with default settings, DynamoDB creates a table that uses on-demand capacity mode instead of provisioned capacity mode.	February 24, 2025
New best practice for configuring your DAX client	Published a new DAX prescriptive guidance best practices topic for configuring your DAX client. For more information, see Configuring your DAX client.	February 17, 2025

New best practices for bulk data operations	Published new best practices topics for working with complex data models in DynamoDB. For more information, see Best practices for using bulk data operations in DynamoDB.	January 9, 2025
Amazon DynamoDB now supports configurable point-in-time recovery (PITR)	DynamoDB now supports configurable recovery periods for PITR. You can now set the PITR period for each table to between 1 and 35 days. For more information, see Point-in-time backups for DynamoDB.	January 7, 2025
Published new topic on integrating Amazon Managed Streaming for Apache Kafka with Amazon DynamoDB	Learn how Amazon Managed Streaming for Apache Kafka integrates with Amazon DynamoDB by reading data from Apache Kafka topics and storing it in DynamoDB. For more information, see Integrating DynamoDB with Amazon Managed Streaming for Apache Kafka.	December 26, 2024
DynamoDB introduces support for a zero-ETL integration with SageMaker AI Lakehouse	DynamoDB introduces a zero-ETL integration that automates the extracting and loading of data from Amazon DynamoDB into a customer's data lake. For more informati on, see DynamoDB zero-ETL integration with SageMaker AI Lakehouse.	December 3, 2024

DynamoDB global tables now supports multi-Region strong consistency	With multi-Region strong consistency, you can build highly available multi-Region applications with a recovery point objective (RPO) of zero, achieving the highest level of resilience. For more information, see Global tables with multi-Region strong consistency.	December 3, 2024
DynamoDB managed policy update	Added two new permissions to the `AmazonDynamoDBReadOnlyAccess` managed policy: `dynamodb:GetAbacStatus` and `dynamodb:UpdateAbacStatus`. These permissions allow you to view the ABAC status and enable ABAC for your AWS account in the current Region. For more information, see AWS managed policy: AmazonDynamoDBReadOnlyAccess.	November 18, 2024

DynamoDB introduces support for attribute-based access control (ABAC)	ABAC is an authorization strategy that lets you define access permissions based on tags attached to users, roles, and AWS resources. ABAC uses tag-based conditions in your AWS Identity and Access Management (IAM) policies or other policies to allow or deny specific actions on your tables or indexes when IAM principals' tags match the tags for the tables. For more information, see Using attribute-based access control with DynamoDB.	November 18, 2024
DynamoDB introduces warm throughput for on-demand and provisioned tables	DynamoDB now supports warm throughput. Warm throughput provides visibilit y into the number of read and write operations your DynamoDB table can instantaneously support as well as the ability to pre-warm your DynamoDB tables. For more information, see DynamoDB warm throughput.	November 13, 2024

Ability to consume Amazon DynamoDB Streams records with Apache Flink	You can now consume Amazon DynamoDB Streams records with Apache Flink and leverage the Amazon Managed Service for Apache Flink to quickly build and manage end-to-end stream processing applications. For more information, see DynamoDB Streams and Apache Flink.	November 12, 2024
Published new billing topics for global tables and backups	Published two new topics regarding billing for global tables and billing for backups. For more information, see Understanding Amazon DynamoDB billing for global tables and Understanding Amazon DynamoDB billing for backups.	October 16, 2024
Amazon DynamoDB zero-ETL integration with Amazon Redshift	Amazon DynamoDB zero-ETL integration with Amazon Redshift provides a no-code, fully managed ETL pipeline with replication from DynamoDB to Amazon Redshift. For more informati on, see Amazon DynamoDB zero-ETL integration with Amazon Redshift.	October 15, 2024

Documentation-only update to add a topic about using generative AI with DynamoDB	Published a new topic that provides information about using generative AI with DynamoDB, including examples of gen AI use cases for DynamoDB. For more information, see Using generative AI with DynamoDB.	October 11, 2024
SDK now supports AWS account-based endpoints	Added documentation for account-based endpoints and the `ACCOUNT_ID_ENDPOINT_MODE` setting for SDK clients. For more information, see SDK support of AWS account-based endpoints.	September 3, 2024
Redesigned the getting started experience	Redesigned the getting started experience to consolidate information and get you started with quicker onboarding. For more information, see Getting started with DynamoDB.	August 1, 2024
DAX expansion to new Regions for Spain and Sweden	DAX is now available in Spain and Sweden Regions. For more information, see DAX cluster components.	July 30, 2024
Restructured and consolidated the DynamoDB backup and restore documentation	The DynamoDB Developer Guide has a new structure for backing up and restoring. For more information, see Backup and restore for DynamoDB.	July 2, 2024

What is Amazon DynamoDB? topic rewrite	Published a revised and updated version of the What is Amazon DynamoDB? topic. For more information, see What is Amazon DynamoDB?.	June 21, 2024
Integrate DynamoDB Streams with EventBridge	Published a new topic on integrating DynamoDB Streams with EventBridge. For more information, see Integrating with EventBridge.	June 21, 2024
DAX prescriptive guidance	Published a new best practices topic that provides you comprehensive insights for using DynamoDB Accelerator effectively. This topic covers performan ce optimization, cost management, and operation al best practices. For more information, see DAX prescriptive guidance.	June 3, 2024
Migrating a DynamoDB table from one account to another	Added a new topic on migrating DynamoDB tables from one account to another. For more information, see Migrating a DynamoDB table from one account to another.	May 29, 2024
Restructured and consolida ted the DynamoDB monitorin g and logging documentation	A new structure for monitorin g and logging in DynamoDB includes three concise chapters for metrics, logging operations, and contributor insights.	May 3, 2024

Restructured and consolidated the DynamoDB capacity mode documentation	DynamoDB guide now includes a new chapter that contains all information about the DynamoDB capacity modes – *on-demand* and *provisioned*. With this update, the *Considerations when changing read/write Capacity Mode* topic has been moved inside the Best practices chapter. This topic is now renamed as *Considerations when switching capacity modes* and includes elaborate information about the best practices when switching between the capacity modes. In addition, the guide now features a new chapter that includes all information about DynamoDB reads and writes, and capacity units consumptions for read and write operations. For more information, see DynamoDB throughput capacity, Considerations when switching capacity modes, and DynamoDB reads and writes.	May 1, 2024

Maximum number of on-demand requests	You can now specify the maximum number of on-demand requests that an individual table, index, or both, can perform. Specifying the maximum on-demand throughput will help keep your table-level usage and costs bounded and protect against inadvertent surge in consumed resources. For more information, see Maximum throughput for on-demand tables.	May 1, 2024
NoSQL Workbench operation builder improvements	NoSQL Workbench now includes native support for dark mode. Improved table and item operations in the operations builder. Item results and operation builder request information is available in JSON format. For more information, see NoSQL Workbench operation builder.	April 24, 2024

Resource-based policies for Amazon DynamoDB resources	DynamoDB now supports resource-based policies for tables, indexes, and streams. Resource-based policies let you define access permissions by specifying who has access to each resource, and the actions they are allowed to perform on each resource. For more information, see Using resource-based policies for DynamoDB.	March 20, 2024
DynamoDB managed policy update	Added a new permission `dynamodb:GetResourcePolicy` to the `AmazonDynamoDBReadOnlyAccess` managed policy. This permission provides access to read resource-based policies attached to DynamoDB resources. For more information, see AWS managed policy: AmazonDynamoDBReadOnlyAccess.	March 20, 2024

AWS PrivateLink for Amazon DynamoDB	Amazon DynamoDB now supports AWS PrivateLink. With AWS PrivateLink, you can simplify private network connectivity between virtual private clouds (VPCs), DynamoDB, and your on-premises data centers using interface VPC endpoints and private IP addresses. For more information, see AWS PrivateLink for DynamoDB.	March 19, 2024
Programming with JavaScript guide	Amazon DynamoDB presents a programming guide for AWS SDK for JavaScript. Learn about the AWS SDK for JavaScript, abstraction layers, configuring connection, handling errors, defining retry policies, managing keep-alive, and more. For more information, see Programming with JavaScript.	March 6, 2024

Programming with AWS SDK for Java 2.x guide	Created a new programming guide that goes in depth about high-level, low-level, and document interfaces, HTTP clients and their configuration, error handling, and addresses the most common configuration settings that you should consider when using the SDK for Java 2.x. For more information, see Programming Amazon DynamoDB with AWS SDK for Java 2.x.	March 5, 2024
Clone tables with NoSQL Workbench	Allow developers to use NoSQL Workbench to copy or clone tables between development environments and regions (DynamoDB Local and DynamoDB web). For more information, see Cloning tables with NoSQL Workbench.	February 26, 2024
Programming with Python guide	Created a new guide that goes in depth about both high level and low level libraries and addresses the most common configuration settings that one should consider when using the Python SDK. For more information, see Programming with Python.	January 5, 2024

Time to live (TTL) topic rewrite	Completely rewrote the TTL section of the guide. The new guide helps you get started with TTL by providing ready-to-use code snippets along the way. The current code snippets provided are in Python and Javascript. For more information, see TTL.	December 20, 2023
Best Practices for Understanding your AWS Billing and Usage Reports	Added a new section that clarifies various usage types and the charges for those usage types in DynamoDB. For more information, see Billing and usage reports.	December 15, 2023
Amazon DynamoDB zero-ETL integration with Amazon OpenSearch Service	Amazon DynamoDB now supports zero-ETL integration with Amazon OpenSearch Service, which lets you perform a search on your DynamoDB data by automatically replicating and transforming it without custom code or infrastructure. For more information, see DynamoDB zero-ETL integration with Amazon OpenSearch Service.	November 28, 2023
Migrating to DynamoDB from a relational database	Created a migration guide to help users understsand how to migrate to DynamoDB from a relational database.	November 27, 2023

Generate sample data with NoSQL Workbench	NoSQL Workbench for Amazon DynamoDB now supports creating data models directly from sample data model templates to help you design data schemas for your workloads. You can use this feature to get familiar with NoSQL data modeling best practices when building your applications on DynamoDB.	September 28, 2023
Incremental Export to S3	You can now export data that was inserted, updated or deleted, in small increments. With incremental export, you can export changed data ranging from a few megabytes to terabytes with a few clicks in the AWS Management Console, an API call, or the AWS Command Line Interface.	September 26, 2023
Data modeling for DynamoDB	You can now learn more about data modeling with DynamoDB examples that focus on specific use cases, their access patterns, and step-by-step guidance in realizing those access patterns.	July 14, 2023

Troubleshooting section	You can now find troubleshooting content for latency and throttling issues that might occur in your DynamoDB tables.	March 13, 2023
Deletion protection for Amazon DynamoDB	Deletion protection is now available for Amazon DynamoDB tables in all AWS Regions. DynamoDB now makes it possible for you to protect your tables from accidental deletion when performing regular table management operations.	March 8, 2023
CloudFormation support for KDSD in global tables	Amazon Kinesis Data Streams for DynamoDB now supports CloudFormation for DynamoDB global tables, which means you can enable streaming to an Amazon Kinesis Data Streams on your DynamoDB global tables with CloudFormation templates.	February 15, 2023
DynamoDB local supports 100 actions per transaction	You can now perform up to 100 actions in a single transaction on DynamoDB local.	February 9, 2023

Using the DynamoDB Well-Architected Lens to optimize your DynamoDB workload	You can now use the DynamoDB Well-Architected Lens, a collection of design principles and guidance that you can use for designing well-architected DynamoDB workloads.	February 3, 2023
PartiQL GovGloud availability	PartiQL—a SQL-compatible query language for Amazon DynamoDB is now supported in AWS GovCloud (US-East) and AWS GovCloud (US-West) .	December 21, 2022
Single installation suite for NoSQL Workbench and DynamoDB local	NoSQL Workbench for DynamoDB now includes a guided DynamoDB local installation process to streamline setting up your DynamoDB local development environment.	December 6, 2022
Bulk import from S3	Amazon DynamoDB now makes it easier for you to migrate and load data into new DynamoDB tables by supporting bulk data imports from Amazon S3.	August 18, 2022

Enhanced integration with Service Quotas	Service Quotas now enables you to proactively manage your account and table quotas. You can view current values, set alarms for when your utilization of a quota exceeds a configurable threshold, and more.	June 15, 2022
NoSQL Workbench adds table and GSI support	You can now use NoSQL Workbench for table and global secondary index (GSI) control plane operations such as CreateTable, UpdateTable, and DeleteTable.	June 2, 2022
Standard-infrequent access table class now available in China	Amazon DynamoDB Standard-Infrequent Access table class is available in China Regions. Reduce your DynamoDB costs by up to 60 percent, by using this new table class for tables that store infrequently accessed data.	April 18, 2022
Increase in default service quotas and table management operations	DynamoDB increased the default quota for the number of tables per account and Region from 256 to 2,500 tables, and increased the number of concurrent table management operations from 50 to 500.	March 9, 2022

Optional limiting of items with PartiQL for DynamoDB	DynamoDB can limit the number of items processed in PartiQL for DynamoDB operations as an optional parameter on each request.	March 8, 2022
AWS Backup integration available in China (Beijing and Ningxia) Regions	AWS Backup now integrates with DynamoDB in the China (Beijing and Ningxia) Regions. You can meet complianc e and business continuity requirements more easily through enhanced backup features in AWS Backup, such as cross-account and cross-Region backups.	January 26, 2022
Throughput capacity information through PartiQL API calls	DynamoDB can return the throughput capacity consumed by PartiQL API calls to help you optimize your queries and throughput costs.	January 18, 2022
AWS Backup integration	DynamoDB now helps you meet compliance and business continuity requireme nts more easily through enhanced backup features in AWS Backup, such as cross-account and cross-Region backups.	November 24, 2021

NoSQL Workbench import/export datasets in CSV	NoSQL Workbench for Amazon DynamoDB now enables you to import and automatically populate sample data to help build and visualize your data models.	October 11, 2021
Filter and retrieve Amazon DynamoDB Streams data-plane activity with AWS CloudTrail	Amazon DynamoDB now provides you more granular control of audit logging by enabling you to filter Streams data-plane API activity in AWS CloudTrail.	September 22, 2021
Updated console	The DynamoDB console is now your default console to help you manage data more easily, simplify your common tasks, and give you faster access to resources and features.	August 25, 2021
DAX SDK For Java 2.x is now available	DynamoDB Accelerator (DAX) SDK for Java 2.x is now available and is compatible with the AWS SDK for Java 2.x. You can benefit from the latest features, including non-blocking I/O.	July 29, 2021
NoSQL Workbench feature updates including control plane operations	NoSQL Workbench for Amazon DynamoDB now helps you run frequent operations more easily to modify and access table data.	July 28, 2021

DynamoDB global tables are now available in the Asia Pacific Region	DynamoDB global tables are now available in the Asia Pacific (Osaka) Region. Replicate your DynamoDB tables automatically across your choice of 22 AWS Regions.	July 28, 2021
DAX is now available in China	DynamoDB Accelerator (DAX) is now available in the China (Beijing) Region, operated by Sinnet.	July 28, 2021
DAX encryption in transit	DynamoDB Accelerator (DAX) now supports encryption in transit of data between your applications and DAX clusters, and between the nodes within a DAX cluster.	July 24, 2021
CloudFormation and CloudTrail integration	Integration with AWS CloudFormation and security enhancements with CloudFormation data-plane logging.	June 18, 2021
CloudFormation now supported for global tables	Amazon DynamoDB global tables now support AWS CloudFormation, which means you can create global tables and manage their settings with CloudFormation templates.	May 14, 2021

Amazon DynamoDB local support for Java 2.x	You now can use the AWS SDK for Java 2.x with DynamoDB local, the downloadable version of Amazon DynamoDB. With DynamoDB local, you can develop and test applicati ons by using a version of DynamoDB running in your local development environme nt without incurring any additional costs.	May 3, 2021
NoSQL Workbench now supports AWS CloudForm ation	NoSQL Workbench for Amazon DynamoDB now supports AWS CloudForm ation, so you can manage and modify DynamoDB data models with CloudFormation templates. In addition, you now can configure table capacity settings in NoSQL Workbench.	April 22, 2021
DynamoDB and AWS Amplify now feature integration	AWS Amplify now orchestra tes multiple DynamoDB global secondary index updates in a single deployment.	April 20, 2021
AWS CloudTrail to log Amazon DynamoDB Streams data-plane APIs	You now can use AWS CloudTrail to log Amazon DynamoDB Streams data-plan e API activity, and monitor and investigate item-level changes in your DynamoDB tables.	April 20, 2021

Amazon Kinesis Data Streams for Amazon DynamoDB now supports AWS CloudForm ation	Amazon Kinesis Data Streams for Amazon DynamoDBnow supports AWS CloudForm ation, which means you can enable streaming to an Amazon Kinesis data stream on your DynamoDB tables with CloudFormation templates. By streaming your DynamoDB data changes to a Kinesis data stream, you can build advanced streaming applications with AAmazon Kinesis services.	April 12, 2021
Amazon Keyspaces now offers FIPS 140-2 compliant endpoints	Amazon Keyspaces (for Apache Cassandra) now offers Federal Information Processing Standards (FIPS) 140-2 compliant endpoints to help you run highly regulated workloads more easily. FIPS 140-2 is a US and Canadian government standard that specifies the security requirements for cryptographic modules that protect sensitive information.	April 8, 2021
Amazon EC2 T3 instances for DAX	DAX now supports Amazon EC2 T3 instance types, which provide a baseline level of CPU performance with the ability to burst above the baseline when needed.	February 15, 2021

NoSQL Workbench for Amazon DynamoDB support for PartiQL	You now can use the NoSQL Workbench for DynamoDB to build PartiQL statements for DynamoDB.	December 4, 2020
PartiQL for DynamoDB	You now can use PartiQL for DynamoDB—a SQL-compatible query language—to interact with DynamoDB tables and run ad hoc queries by using the AWS Management Console, AWS Command Line Interface, and DynamoDB APIs for PartiQL.	November 23, 2020
Amazon Kinesis Data Streams for Amazon DynamoDB	You now can use Amazon Kinesis Data Streams for Amazon DynamoDB with your DynamoDB tables to capture item-level changes and replicate them to a Kinesis data stream.	November 23, 2020
DynamoDB table export	You can now export your DynamoDB tables to Amazon S3, enabling you to perform analytics and complex queries on your data with services like Athena, AWS Glue, and Lake Formation.	November 9, 2020

Support for empty values	DynamoDB now supports empty values for non-key String and Binary attributes in DynamoDB tables. Empty value support gives you greater flexibility to use attributes for a broader set of use cases without having to transform such attribute s before sending them to DynamoDB. List, Map, and Set data types also support empty String and Binary values.	May 18, 2020
NoSQL Workbench for Amazon DynamoDB support for Linux	NoSQL Workbench for Amazon DynamoDB is now supported on Linux- Ubuntu , Fedora and Debian.	May 4, 2020
CloudWatch Contributor Insights for DynamoDB – GA	CloudWatch Contributor Insights for DynamoDB is generally available. CloudWatch Contributor Insights for DynamoDB is a diagnostic tool that provides an at-a-glance view of your DynamoDB table's traffic trends and helps you identify your table's most frequently accessed keys (also known as hot keys).	April 2, 2020

Upgrading global tables	You now can update your global tables from version 2017.11.29 to the latest version of global tables (2019.11.21), with a few clicks in the DynamoDB Console. By upgrading the version of your global tables, you can increase the availability of your DynamoDB tables easily by extending your existing tables into additiona l AWS Regions, with no table rebuilds required.	March 16, 2020
NoSQL Workbench for Amazon DynamoDB – GA	NoSQL Workbench for Amazon DynamoDB is generally available. Use the NoSQL Workbench to design, create, query, and manage DynamoDB tables.	March 2, 2020
DAX cache cluster metrics	DAX support for new CloudWatch metrics, that allow you to better understand your DAX cluster's performance.	February 6, 2020
CloudWatch Contributor Insights for DynamoDB – Preview	CloudWatch Contributor Insights for DynamoDB is a diagnostic tool that provides an at-a-glance view of your DynamoDB table's traffic trends and helps you identify your table's most frequently accessed keys (also known as hot keys).	November 26, 2019

Adaptive capacity support for imbalanced workload	Amazon DynamoDB adaptive capacity now handles imbalanced workloads better by isolating frequently accessed items automatic ally. If your application drives disproportionately high traffic to one or more items, DynamoDB will rebalance your partitions such that frequently accessed items do not reside on the same partition.	November 26, 2019
Support for customer managed keys	DynamoDB now supports customer managed keys, which means you can have full control over how you encrypt and manage the security of your DynamoDB data.	November 25, 2019
NoSQL Workbench support for DynamoDB local (Downloadable Version)	The NoSQL Workbench now supports connecting to DynamoDB local (Download able Version) to design, create, query, and manage DynamoDB tables.	November 8, 2019

NoSQL Workbench - Preview	This is the initial release of NoSQL Workbench for DynamoDB. Use NoSQL Workbench to design, create, query, and manage DynamoDB tables. For more information, see NoSQL Workbench for Amazon DynamoDB (Preview).	September 16, 2019
DAX adds support for transactional operations using Python and .NET	DAX supports the `TransactWriteItems` and `TransactGetItems` APIs for applications written in Go, Java, .NET, Node.js, and Python. For more information, see In-Memory Acceleration with DAX.	February 14, 2019
Amazon DynamoDB local (Downloadable Version) Updates	DynamoDB local (Downloadable Version) now supports transactional APIs, on-demand read/write capacity, capacity reporting for read and write operations, and 20 global secondary indexes. For more information, see Differences Between Downloadable DynamoDB and the DynamoDB Web Service.	February 4, 2019

Amazon DynamoDB On-Demand	DynamoDB on-demand is a flexible billing option capable of serving thousands of requests per second without capacity planning. DynamoDB on-demand offers pay-per-request pricing for read and write requests so that you pay only for what you use. For more information, see DynamoDB throughput capacity.	November 28, 2018
Amazon DynamoDB Transactions	DynamoDB transactions make coordinated, all-or-nothing changes to multiple items both within and across tables, providing atomicity, consistency, isolation, and durability (ACID) in DynamoDB. For more information, see Amazon DynamoDB Transactions.	November 27, 2018

Amazon DynamoDB encrypts all customer data at rest	DynamoDB encryption at rest provides an additional layer of data protection by securing your data in the encrypted table, including its primary key, local and global secondary indexes, streams, global tables, backups, and DAX clusters whenever the data is stored in durable media. For more informati on, see Amazon DynamoDB Encryption at Rest.	November 15, 2018
Use Amazon DynamoDB Local More Easily with the New Docker Image	Now, it's easier to use DynamoDB local, the downloadable version of DynamoDB, to help you develop and test your DynamoDB applications by using the new DynamoDB local Docker image. For more information, see DynamoDB (Downloadable Version) and Docker.	August 22, 2018

DynamoDB Accelerator (DAX) Adds Support for Encryption at Rest	DynamoDB Accelerator (DAX) now supports encryption at rest for new DAX clusters to help you accelerate reads from Amazon DynamoDB tables in security-sensitive applications that are subject to strict compliance and regulatory requirements. For more information, see DAX Encryption at Rest.	August 9, 2018
DynamoDB point-in-time recovery (PITR) adds support for restoring deleted tables	If you delete a table with point-in-time recovery enabled, a system backup is automatically created and is retained for 35 days (at no additional cost). For more information, see Before You Begin Using Point In Time Recovery.	August 7, 2018
Updates now available over RSS	You can now subscribe to the RSS feed (at the top left corner of this page) to receive notifications about updates to the Amazon DynamoDB Developer Guide.	July 3, 2018

Earlier updates

The following table describes important changes of the *DynamoDB Developer Guide* before July 3, 2018.

Change	Description	Date Changed
Go support for DAX	Now, you can enable microsecond read performance for Amazon DynamoDB tables in your applications written in the Go programming language by using the new DynamoDB Accelerator (DAX) SDK for Go. For more information, see DAX SDK for Go.	June 26, 2018
DynamoDB announces SLA	DynamoDB has released a public availability SLA. For more information, see Amazon DynamoDB Service Level Agreement.	June 19, 2018
DynamoDB continuous backups and Point-In-Time Recovery (PITR)	Point-in-time recovery helps protect your Amazon DynamoDB tables from accidental write or delete operations. With point in time recovery, you don't have to worry about creating, maintaining, or scheduling on-demand backups. For example, suppose that a test script writes accidentally to a production DynamoDB table. With point-in-time recovery, you can restore that table to any point in time during the last 35 days. DynamoDB maintains incremental backups of your	April 25, 2018

Change	Description	Date Changed
	table. For more information, see Point-in-time backups for DynamoDB.	
Encryption at rest for DynamoDB	DynamoDB encryption at rest, available for new DynamoDB tables, helps you secure your application data in Amazon DynamoDB tables by using AWS-managed encryption keys stored in AWS Key Management Service. For more information, see DynamoDB encryption at rest.	February 8, 2018
DynamoDB Backup and restore	On-Demand Backup allows you to create full backups of your DynamoDB tables data for data archival, helping you meet your corporate and governmental regulatory requirements. You can backup tables from a few megabytes to hundreds of terabytes of data, with no impact on performance and availability to your production applications. For more information, see Backup and restore for DynamoDB.	November 29, 2017

Change	Description	Date Changed
DynamoDB Global tables	Global Tables builds upon DynamoDB's global footprint to provide you with a fully managed, multi-region, and multi-active database that provides fast, local, read and write performance for massively scaled, global applications. Global Tables replicates your Amazon DynamoDB tables automatic ally across your choice of AWS regions. For more informati on, see Global tables - multi-active, multi-Region replicati on.	November 29, 2017
Node.js support for DAX	Node.js developers can leverage DynamoDB Accelerat or (DAX), using the DAX client for Node.js. For more information, see In-memory acceleration with DynamoDB Accelerator (DAX).	October 5, 2017

Change	Description	Date Changed
VPC Endpoints for DynamoDB	DynamoDB endpoints allow Amazon EC2 instances in your Amazon VPC to access DynamoDB, without exposure to the public Internet. Network traffic between your VPC and DynamoDB does not leave the Amazon network. For more information, see Using Amazon VPC endpoints to access DynamoDB.	August 16, 2017
Auto Scaling for DynamoDB	DynamoDB auto scaling eliminates the need for manually defining or adjust provisioned throughput settings. Instead, DynamoDB auto scaling dynamical ly adjusts read and write capacity in response to actual traffic patterns. This allows a table or a global secondary index to increase its provision ed read and write capacity to handle sudden increases in traffic, without throttling. When the workload decreases , DynamoDB auto scaling decreases the provisioned capacity. For more informati on, see Managing throughpu t capacity automatically with DynamoDB auto scaling.	June 14, 2017

Change	Description	Date Changed
DynamoDB Accelerator (DAX)	DynamoDB Accelerator (DAX) is a fully managed, highly available, in-memory cache for DynamoDB that delivers up to a 10x performance improvement – from milliseconds to microseconds – even at millions of requests per second. For more information, see In-memory acceleration with DynamoDB Accelerator (DAX).	April 19, 2017
DynamoDB now supports automatic item expiration with Time to Live (TTL)	Amazon DynamoDB Time to Live (TTL) enables you to automatically delete expired items from your tables, at no additional cost. For more information, see Using time to live (TTL) in DynamoDB.	Feb 27, 2017
DynamoDB now supports Cost Allocation Tags	You can now add tags to your Amazon DynamoDB tables for improved usage categorization and more granular cost reporting. For more information, see Adding tags and labels to resources in DynamoDB.	Jan 19, 2017

Change	Description	Date Changed
New DynamoDB `DescribeLimits` API	The `DescribeLimits` API returns the current provisioned capacity limits for your AWS account in a region, both for the region as a whole and for any one DynamoDB table that you create there. It lets you determine what your current account-level limits are so that you can compare them to the provisioned capacity that you are currently using, and have plenty of time to apply for an increase before you hit a limit. For more information, see Quotas in Amazon DynamoDB and the DescribeLimits in the *Amazon DynamoDB API Reference*.	March 1, 2016

Change	Description	Date Changed
DynamoDB Console Update and New Terminology for Primary Key Attributes	The DynamoDB management console has been redesigned to be more intuitive and easy to use. As part of this update, we are introducing new terminology for primary key attributes: • **Partition Key—**also known as a *hash attribute*. • **Sort Key—**also known as a *range attribute*. Only the names have changed; the functionality remains the same. When you create a table or a secondary index, you can choose either a simple primary key (partition key only), or a composite primary key (partition key and sort key). The DynamoDB documentation has been updated to reflect these changes.	November 12, 2015

Change	Description	Date Changed
Amazon DynamoDB Storage Backend for Titan	The DynamoDB Storage Backend for Titan is a storage backend for the Titan graph database implemented on top of Amazon DynamoDB. When using the DynamoDB Storage Backend for Titan, your data benefits from the protection of DynamoDB, which runs across Amazon's high-availability data centers. The plugin is available for Titan version 0.4.4 (primarily for compatibility with existing applications) and Titan version 0.5.4 (recommended for new applications). Like other storage backends for Titan, this plugin supports the Tinkerpop stack (versions 2.4 and 2.5), including the Blueprints API and the Gremlin shell.	August 20, 2015

Change	Description	Date Changed
DynamoDB Streams, Cross-Region Replication, and Scan with Strongly Consistent Reads	DynamoDB Streams captures a time-ordered sequence of item-level modifications in any DynamoDB table, and stores this information in a log for up to 24 hours. Applications can access this log and view the data items as they appeared before and after they were modified, in near real time. For more information, see Change data capture for DynamoDB Streams and the DynamoDB Streams API Reference. DynamoDB cross-region replication is a client-side solution for maintaining identical copies of DynamoDB tables across different AWS regions, in near real time. You can use cross region replicati on to back up DynamoDB tables, or to provide low-laten cy access to data where users are geographically distribut ed. The DynamoDB `Scan` operation uses eventuall y consistent reads, by default. You can use strongly consistent reads instead by setting the `Consisten`	July 16, 2015

Change	Description	Date Changed
	`tRead` parameter to true. For more information, see Read consistency for scan and Scan in the Amazon DynamoDB API Reference.	
AWS CloudTrail support for Amazon DynamoDB	DynamoDB is now integrated with CloudTrail. CloudTrail captures API calls made from the DynamoDB console or from the DynamoDB API and tracks them in log files. For more information, see Logging DynamoDB operations by using AWS CloudTrail and the AWS CloudTrail User Guide.	May 28, 2015

Change	Description	Date Changed
Improved support for Query expressions	This release adds a new `KeyConditionExpression` parameter to the `Query` API. A `Query` reads items from a table or an index using primary key values. The `KeyConditionExpression` parameter is a string that identifies primary key names, and conditions to be applied to the key values; the `Query` retrieves only those items that satisfy the expression. The syntax of `KeyConditionExpression` is similar to that of other expression parameters in DynamoDB, and allows you to define substitution variables for names and values within the expression. For more information, see Querying tables in DynamoDB.	April 27, 2015

Change	Description	Date Changed
New comparison functions for conditional writes	In DynamoDB, the `ConditionExpression` parameter determines whether a `PutItem`, `UpdateItem`, or `DeleteItem` succeeds: The item is written only if the condition evaluates to true. This release adds two new functions, `attribute_type` and `size`, for use with `ConditionExpression`. These functions allow you to perform conditional writes based on the data type or size of an attribute in a table. For more information, see DynamoDB condition expression CLI example.	April 27, 2015

Change	Description	Date Changed
Scan API for secondary indexes	In DynamoDB, a `Scan` operation reads all of the items in a table, applies user-defined filtering criteria, and returns the selected data items to the applicati on. This same capability is now available for secondary indexes too. To scan a local secondary index or a global secondary index, you specify the index name and the name of its parent table. By default, an index `Scan` returns all of the data in the index; you can use a filter expression to narrow the results that are returned to the application. For more information, see Scanning tables in DynamoDB.	February 10, 2015

Change	Description	Date Changed
Online operations for global secondary indexes	Online indexing lets you add or remove global secondary indexes on existing tables. With online indexing, you do not need to define all of a table's indexes when you create a table; instead, you can add a new index at any time. Similarly, if you decide you no longer need an index, you can remove it at any time. Online indexing operation s are non-blocking, so that the table remains available for read and write activity while indexes are being added or removed. For more information, see Managing Global Secondary Indexes in DynamoDB.	January 27, 2015

Change	Description	Date Changed
Document model support with JSON	DynamoDB allows you to store and retrieve documents with full support for document models. New data types are fully compatible with the JSON standard and allow you to nest document elements within one another. You can use document path dereference operators to read and write individual elements, without having to retrieve the entire document. This release also introduces new expressio n parameters for specifying projections, conditions and update actions when reading or writing data items. To learn more about document model support with JSON, see Data types and Using expressions in DynamoDB.	October 7, 2014
Flexible scaling	For tables and global secondary indexes, you can increase provisioned read and write throughput capacity by any amount, provided that you stay within your per-table and per-account limits. For more information, see Quotas in Amazon DynamoDB.	October 7, 2014

Change	Description	Date Changed
Larger item sizes	The maximum item size in DynamoDB has increased from 64 KB to 400 KB. For more information, see Quotas in Amazon DynamoDB.	October 7, 2014
Improved conditional expressions	DynamoDB expands the operators that are available for conditional expressio ns, giving you additional flexibility for conditional puts, updates, and deletes. The newly available operators let you check whether an attribute does or does not exist, is greater than or less than a particular value, is between two values, begins with certain characters, and much more. DynamoDB also provides an optional *OR* operator for evaluatin g multiple conditions. By default, multiple conditions in an expression are *AND*ed together, so the expression is true only if all of its condition s are true. If you specify *OR* instead, the expression is true if one or more one condition s are true. For more informati on, see Working with items and attributes in DynamoDB.	April 24, 2014

Change	Description	Date Changed
Query filter	The DynamoDB `Query` API supports a new `QueryFilter` option. By default, a `Query` finds items that match a specific partition key value and an optional sort key condition. A `Query` filter applies conditional expressions to other, non-key attributes; if a `Query` filter is present, then items that do not match the filter conditions are discarded before the `Query` results are returned to the application. For more information, see Querying tables in DynamoDB.	April 24, 2014

Change	Description	Date Changed
Data export and import using the AWS Management Console	The DynamoDB console has been enhanced to simplify exports and imports of data in DynamoDB tables. With just a few clicks, you can set up an AWS Data Pipeline to orchestrate the workflow, and an Amazon Elastic MapReduce cluster to copy data from DynamoDB tables to an Amazon S3 bucket, or vice-versa. You can perform an export or import one time only, or set up a daily export job. You can even perform cross-region exports and imports, copying DynamoDB data from a table in one AWS region to a table in another AWS region.	March 6, 2014
Reorganized higher-level API documentation	Information about the following APIs is now easier to find: • Java: DynamoDBMappper • .NET: Document model and object-persistence model These higher-level APIs are now documented here: Higher-level programming interfaces for DynamoDB.	January 20, 2014

Change	Description	Date Changed
Global secondary indexes	DynamoDB adds support for global secondary indexes. As with a local secondary index, you define a global secondary index by using an alternate key from a table and then issuing Query requests on the index. Unlike a local secondary index, the partition key for the global secondary index does not have to be the same as that of the table; it can be any scalar attribute from the table. The sort key is optional and can also be any scalar table attribute . A global secondary index also has its own provisioned throughput settings, which are separate from those of the parent table. For more information, see Improving data access with secondary indexes in DynamoDB and Using Global Secondary Indexes in DynamoDB.	December 12, 2013

Change	Description	Date Changed
Fine-grained access control	DynamoDB adds support for fine-grained access control. This feature allows customers to specify which principals (users, groups, or roles) can access individual items and attributes in a DynamoDB table or secondary index. Applications can also leverage web identity federation to offload the task of user authentication to a third-par ty identity provider, such as Facebook, Google, or Login with Amazon. In this way, applications (including mobile apps) can handle very large numbers of users, while ensuring that no one can access DynamoDB data items unless they are authorized to do so. For more informati on, see Using IAM policy conditions for fine-grained access control.	October 29, 2013

Change	Description	Date Changed
4 KB read capacity unit size	The capacity unit size for reads has increased from 1 KB to 4 KB. This enhancement can reduce the number of provisioned read capacity units required for many applications. For example, prior to this release, reading a 10 KB item would consume 10 read capacity units; now that same 10 KB read would consume only 3 units (10 KB / 4 KB, rounded up to the next 4 KB boundary). For more information, see DynamoDB throughput capacity.	May 14, 2013
Parallel scans	DynamoDB adds support for parallel Scan operations. Applications can now divide a table into logical segments and scan all of the segments simultaneously. This feature reduces the time required for a Scan to complete, and fully utilizes a table's provisioned read capacity. For more information, see Scanning tables in DynamoDB.	May 14, 2013

Change	Description	Date Changed
Local secondary indexes	DynamoDB adds support for local secondary indexes. You can define sort key indexes on non-key attributes, and then use these indexes in Query requests. With local secondary indexes, applications can efficiently retrieve data items across multiple dimensions. For more information, see Local secondary indexes.	April 18, 2013
New API version	With this release, DynamoDB introduces a new API version (2012-08-10). The previous API version (2011-12-05) is still supported for backward compatibility with existing applications. New applications should use the new API version 2012-08-10. We recommend that you migrate your existing applications to API version 2012-08-10, since new DynamoDB features (such as local secondary indexes) will not be backported to the previous API version. For more information on API version 2012-08-10, see the Amazon DynamoDB API Reference.	April 18, 2013

Change	Description	Date Changed
IAM policy variable support	The IAM access policy language now supports variables. When a policy is evaluated, any policy variables are replaced with values that are supplied by context-based informati on from the authenticated user's session. You can use policy variables to define general purpose policies without explicitly listing all the components of the policy. For more information about policy variables, go to Policy Variables in the *AWS Identity and Access Management Using IAM* guide. For examples of policy variables in DynamoDB, see Identity and Access Management for Amazon DynamoDB.	April 4, 2013
PHP code examples updated for AWS SDK for PHP version 2	Version 2 of the AWS SDK for PHP is now available . The PHP code examples in the Amazon DynamoDB Developer Guide have been updated to use this new SDK. For more information on Version 2 of the SDK, see AWS SDK for PHP.	January 23, 2013

Change	Description	Date Changed
New endpoint	DynamoDB expands to the AWS GovCloud (US-West) region. For the current list of service endpoints and protocols, see Regions and Endpoints.	December 3, 2012
New endpoint	DynamoDB expands to the South America (São Paulo) region. For the current list of supported endpoints, see Regions and Endpoints.	December 3, 2012
New endpoint	DynamoDB expands to the Asia Pacific (Sydney) region. For the current list of supported endpoints, see Regions and Endpoints.	November 13, 2012

Change	Description	Date Changed
DynamoDB implements support for CRC32 checksums , supports strongly consisten t batch gets, and removes restrictions on concurrent table updates.	• DynamoDB calculates a CRC32 checksum of the HTTP payload and returns this checksum in a new header, `x-amz-crc32` . For more information, see DynamoDB low-level API. • By default, read operation s performed by the `BatchGetItem` API are eventually consistent. A new `ConsistentRead` parameter in `BatchGetI tem` lets you choose strong read consistency instead, for any table(s) in the request. For more information, see Descripti on. • This release removes some restrictions when updating many tables simultane ously. The total number of tables that can be updated at once is still 10; however, these tables can now be any combination of `CREATING`, `UPDATING` or `DELETING` status. Additiona lly, there is no longer any minimum amount for increasing or reducing the *ReadCapacityUnits* or *WriteCapacityUnits* for a	November 2, 2012

Change	Description	Date Changed
	table. For more informati on, see Quotas in Amazon DynamoDB.	
Best practices documentation	The Amazon DynamoDB Developer Guide identifies best practices for working with tables and items, along with recommendations for query and scan operations.	September 28, 2012

Change	Description	Date Changed
Support for binary data type	In addition to the Number and String types, DynamoDB now supports Binary data type. Prior to this release, to store binary data, you converted your binary data into string format and stored it in DynamoDB. In addition to the required conversion work on the client-side, the conversio n often increased the size of the data item requiring more storage and potential ly additional provisioned throughput capacity. With the binary type attribute s you can now store any binary data, for example compressed data, encrypted data, and images. For more information see Data types. For working examples of handling binary type data using the AWS SDKs, see the following sections: • Example: Handling binary type attributes using the AWS SDK for Java document API • Example: Handling binary type attributes using the	August 21, 2012

Change	Description	Date Changed
	AWS SDK for .NET low-level API For the added binary data type support in the AWS SDKs, you will need to download the latest SDKs and you might also need to update any existing applications. For information about downloading the AWS SDKs, see .NET code examples.	
DynamoDB table items can be updated and copied using the DynamoDB console	DynamoDB users can now update and copy table items using the DynamoDB Console, in addition to being able to add and delete items. This new functionality simplifies making changes to individual items through the Console.	August 14, 2012
DynamoDB lowers minimum table throughput requirements	DynamoDB now supports lower minimum table throughput requirements, specifically 1 write capacity unit and 1 read capacity unit. For more information, see the Quotas in Amazon DynamoDB topic in the Amazon DynamoDB Developer Guide.	August 9, 2012

Change	Description	Date Changed
Signature Version 4 support	DynamoDB now supports Signature Version 4 for authenticating requests.	July 5, 2012
Table explorer support in DynamoDB Console	The DynamoDB Console now supports a table explorer that enables you to browse and query the data in your tables. You can also insert new items or delete existing items. The `SampleData` and Using the console sections have been updated for these features.	May 22, 2012
New endpoints	DynamoDB availability expands with new endpoints in the US West (N. Californi a) region, US West (Oregon) region, and the Asia Pacific (Singapore) region. For the current list of supported endpoints, go to Regions and Endpoints.	April 24, 2012

Change	Description	Date Changed
BatchWriteItem API support	DynamoDB now supports a batch write API that enables you to put and delete several items from one or more tables in a single API call. For more information about the DynamoDB batch write API, see BatchWriteItem. For information about working with items and using batch write feature using AWS SDKs, see Working with items and attributes in DynamoDB and .NET code examples.	April 19, 2012
Documented more error codes	For more information, see Error handling with DynamoDB.	April 5, 2012
New endpoint	DynamoDB expands to the Asia Pacific (Tokyo) region. For the current list of supported endpoints, see Regions and Endpoints.	February 29, 2012
`ReturnedItemCount` metric added	A new metric, `ReturnedItemCount` , provides the number of items returned in the response of a Query or Scan operation for DynamoDB is available for monitoring through CloudWatch.	February 24, 2012

Change	Description	Date Changed
Added examples for incrementing values	DynamoDB supports incrementing and decrementing existing numeric values. Examples show adding to existing values in the "Updating an Item" sections at: Working with items: Java. Working with items: .NET.	January 25, 2012
Initial product release	DynamoDB is introduced as a new service in Beta release.	January 18, 2012

Legacy features of DynamoDB

The following topics are legacy features that DynamoDB still supports. No active development is made on these features.

Topics

- Global tables version 2017.11.29 (Legacy)
- Previous low-level DynamoDB API version (2011-12-05)
- Legacy DynamoDB conditional parameters

Global tables version 2017.11.29 (Legacy)

Important

This documentation is for version 2017.11.29 (Legacy) of global tables, which should be avoided for new global tables. Customers should use Global Tables version 2019.11.21 (Current) when possible, as it provides greater flexibility, higher efficiency and consumes less write capacity than 2017.11.29 (Legacy).
To determine which version you are using, see Determining the version of a global table.
To update existing global tables from version 2017.11.29 (Legacy) to version 2019.11.21 (Current), see DynamoDB global tables versions.

Topics

- Global tables: How it works
- Best practices and requirements for managing global tables
- Creating a global table (Version 2017.11.29)
- Monitoring global tables
- Using IAM with global tables

Global tables: How it works

> **Important**
>
> This documentation is for version 2017.11.29 (Legacy) of global tables, which should be avoided for new global tables. Customers should use Global Tables version 2019.11.21 (Current) when possible, as it provides greater flexibility, higher efficiency and consumes less write capacity than 2017.11.29 (Legacy).
> To determine which version you are using, see Determining the version of a global table.
> To update existing global tables from version 2017.11.29 (Legacy) to version 2019.11.21 (Current), see DynamoDB global tables versions.

The following sections help you understand the concepts and behavior of global tables in Amazon DynamoDB.

Global table concepts for Version 2017.11.29 (Legacy)

A *global table* is a collection of one or more replica tables, all owned by a single AWS account.

A *replica table* (or *replica*, for short) is a single DynamoDB table that functions as a part of a global table. Each replica stores the same set of data items. Any given global table can only have one replica table per AWS Region.

The following is a conceptual overview of how a global table is created.

1. Create an ordinary DynamoDB table, with DynamoDB Streams enabled, in an AWS Region.
2. Repeat step 1 for every other Region where you want to replicate your data.
3. Define a DynamoDB global table based on the tables that you have created.

The AWS Management Console automates these tasks, so you can create a global table more quickly and easily. For more information, see Creating a global table (Version 2017.11.29).

The resulting DynamoDB global table consists of multiple replica tables, one per Region, that DynamoDB treats as a single unit. Every replica has the same table name and the same primary key schema. When an application writes data to a replica table in one Region, DynamoDB automatically propagates the write to the other replica tables in the other AWS Regions.

Important

To keep your table data in sync, global tables automatically create the following attributes for every item:

- `aws:rep:deleting`
- `aws:rep:updatetime`
- `aws:rep:updateregion`

Do not modify these attributes or create attributes with the same name.

You can add replica tables to the global table so that it can be available in additional Regions. (To do this, the global table must be empty. In other words, none of the replica tables can have any data in them.)

You can also remove a replica table from a global table. If you do this, the table is completely disassociated from the global table. This newly independent table no longer interacts with the global table, and data is no longer propagated to or from the global table.

Warning

Be aware that removing a replica isn't an atomic process. To ensure consistent behavior and known state, you may want to consider diverting your application write traffic away from the replica to be removed ahead of time. After removing it, wait until all replica Region endpoints show the replica as disassociated before making any further writes to it as its own isolated regional table.

Common tasks

Common tasks for global tables work as follows.

You can delete a global table's replica table the same as a regular table. This will stop replication to that Region and delete the table copy kept in that Region. You cannot sever the replication and have copies of the table exist as independent entities.

> **Note**
>
> You won't be able to delete a source table until at least 24 hours after it's used to initiate a new Region. If you try to delete it too soon you will receive an error.

Conflicts can arise if applications update the same item in different Regions at about the same time. To help ensure eventual consistency, DynamoDB global tables use a "last writer wins" method to reconcile between concurrent updates. All the replicas will agree on the latest update and converge toward a state in which they all have identical data.

> **Note**
>
> There are several ways to avoid conflicts, including:
>
> - Using an IAM policy to only allow writes to the table in one Region.
> - Using an IAM policy to route users to only one Region and keeping the other as an idle standby, or alternately routing odd users to one Region and even users to another Region.
> - Avoiding the use of non-idempotent updates such as Bookmark = Bookmark + 1, in favor of static updates such as Bookmark=25.

Monitoring global tables

You can use CloudWatch to observe the metric `ReplicationLatency`. This metric tracks the elapsed time between when an updated item appears in the DynamoDB stream for one replica table, and when that item appears in another replica in the global table. `ReplicationLatency` is expressed in milliseconds and is emitted for every source-Region and destination-Region pair. This is the only CloudWatch metric provided by Global Tables v2.

The latencies you will observe depend on the distance between your chosen Regions, as well as other variables. Latencies in the 0.5 to 2.5 second range for Regions can be common within the same geographic area.

Time To Live (TTL)

You can use Time To Live (TTL) to specify an attribute name whose value indicates the time of expiration for the item. This value is specified as a number in seconds since the start of the Unix epoch.

With global tables legacy version, the TTL deletes are not automatically replicated across other replicas. When an item is deleted via a TTL rule, that work is performed without consuming Write Units.

Be aware that if the source and target table have very low Provisioned write capacity, this may cause throttling as the TTL deletes require write capacity.

Streams and transactions with global tables

Each global table produces an independent stream based on all its writes, regardless of the origination point for those writes. You can choose to consume this DynamoDB stream in one Region or in all Regions independently.

If you want processed local writes but not replicated writes, you can add your own region attribute to each item. Then you can use a Lambda event filter to invoke only the Lambda for writes in the local Region.

Transactional operations provide ACID (Atomicity, Consistency, Isolation, and Durability) guarantees ONLY within the Region where the write is originally made. Transactions aren't supported across Regions in global tables.

For example, if you have a global table with replicas in the US East (Ohio) and US West (Oregon) Regions and perform a TransactWriteItems operation in the US East (Ohio) Region, you may observe partially completed transactions in US West (Oregon) Region as changes are replicated. Changes will only be replicated to other Regions once they have been committed in the source Region.

Note

- Global tables “write around” DynamoDB Accelerator by updating DynamoDB directly. As a result DAX will not be aware it is holding stale data. The DAX cache will only be refreshed when the cache’s TTL expires.
- Tags on global tables do not automatically propagate.

Read and write throughput

Global tables manage read and write throughput in the following ways.

- The write capacity must be the same on all table instances across Regions.
- With Version 2019.11.21 (Current, if the table is set to support auto scaling or is in on-demand mode then the write capacity is automatically kept in sync. The current amount of write capacity provisioned in each Region will rise and fall independently within those synchronized auto scaling settings. If the table is placed in on-demand mode, that mode will sync to the other replicas.
- Read capacity can differ between Regions because reads may not be equal. When adding a global replica to a table, the capacity of the source Region is propagated. After creation you can adjust the read capacity for one replica, and this new setting isn't transferred to the other side.

Consistency and conflict resolution

Any changes made to any item in any replica table are replicated to all the other replicas within the same global table. In a global table, a newly written item is usually propagated to all replica tables within seconds.

With a global table, each replica table stores the same set of data items. DynamoDB does not support partial replication of only some of the items.

An application can read and write data to any replica table. DynamoDB supports eventually consistent reads across Regions, but does not support strongly consistent reads across Regions. If your application only uses eventually consistent reads and only issues reads against one AWS Region, it will work without any modification. However, if your application requires strongly consistent reads then it must perform all the strongly consistent reads and writes in the same Region. Otherwise, if you write to one Region and read from another Region then the read response might include stale data that doesn't reflect the results of recently completed writes in the other Region.

Conflicts can arise if applications update the same item in different Regions at about the same time. To help ensure eventual consistency, DynamoDB global tables use a *last writer wins* reconciliation between concurrent updates, in which DynamoDB makes a best effort to determine the last writer. With this conflict resolution mechanism, all the replicas will agree on the latest update and converge toward a state in which they all have identical data.

Availability and durability

If a single AWS Region becomes isolated or degraded, your application can redirect to a different Region and perform reads and writes against a different replica table. You can apply custom business logic to determine when to redirect requests to other Regions.

If a Region becomes isolated or degraded, DynamoDB keeps track of any writes that have been performed but have not yet been propagated to all of the replica tables. When the Region comes back online, DynamoDB resumes propagating any pending writes from that Region to the replica tables in other Regions. It also resumes propagating writes from other replica tables to the Region that is now back online. All previously successful writes will be propagated eventually no matter how long the Region is isolated.

Best practices and requirements for managing global tables

> **Important**
>
> This documentation is for version 2017.11.29 (Legacy) of global tables, which should be avoided for new global tables. Customers should use Global Tables version 2019.11.21 (Current) when possible, as it provides greater flexibility, higher efficiency and consumes less write capacity than 2017.11.29 (Legacy).
> To determine which version you are using, see Determining the version of a global table.
> To update existing global tables from version 2017.11.29 (Legacy) to version 2019.11.21 (Current), see DynamoDB global tables versions.

Using Amazon DynamoDB global tables, you can replicate your table data across AWS Regions. It is important that the replica tables and secondary indexes in your global table have identical write capacity settings to ensure proper replication of data.

Topics

- Global tables version
- Requirements for adding a new replica table
- Best practices and requirements for managing capacity

Global tables version

There are two versions of DynamoDB global tables available: Global Tables version 2019.11.21 (Current) and Global tables version 2017.11.29 (Legacy). Customers should use Global Tables version 2019.11.21 (Current) when possible, as it provides greater flexibility, higher efficiency and consumes less write capacity than 2017.11.29 (Legacy).

To determine which version you are using, see Determining the version of a global table. To update existing global tables from Version 2017.11.29 (Legacy) to Version 2019.11.21 (Current), see Upgrading global tables.

Requirements for adding a new replica table

If you want to add a new replica table to a global table, each of the following conditions must be true:

- The table must have the same partition key as all of the other replicas.
- The table must have the same write capacity management settings specified.
- The table must have the same name as all of the other replicas.
- The table must have DynamoDB Streams enabled, with the stream containing both the new and the old images of the item.
- None of the new or existing replica tables in the global table can contain any data.

If global secondary indexes are specified, the following conditions must also be met:

- The global secondary indexes must have the same name.
- The global secondary indexes must have the same partition key and sort key (if present).

> **Important**
>
> Write capacity settings should be set consistently across all of your global tables' replica tables and matching secondary indexes. To update write capacity settings for your global table, we strongly recommend using the DynamoDB console or the `UpdateGlobalTableSettings` API operation. `UpdateGlobalTableSettings` applies changes to write capacity settings to all replica tables and matching secondary indexes in a global table automatically. If you use the `UpdateTable`, `RegisterScalableTarget`,

or `PutScalingPolicy` operations, you should apply the change to each replica table and matching secondary index individually. For more information, see UpdateGlobalTableSettings in the Amazon DynamoDB API Reference.
We strongly recommend that you enable auto scaling to manage provisioned write capacity settings. If you prefer to manage write capacity settings manually, you should provision equal replicated write capacity units to all of your replica tables. Also provision equal replicated write capacity units to matching secondary indexes across your global table.
You must also have appropriate AWS Identity and Access Management (IAM) permissions. For more information, see Using IAM with global tables.

Best practices and requirements for managing capacity

Consider the following when managing capacity settings for replica tables in DynamoDB.

Using DynamoDB auto scaling

Using DynamoDB auto scaling is the recommended way to manage throughput capacity settings for replica tables that use the provisioned mode. DynamoDB auto scaling automatically adjusts read capacity units (RCUs) and write capacity units (WCUs) for each replica table based upon your actual application workload. For more information, see Managing throughput capacity automatically with DynamoDB auto scaling.

If you create your replica tables using the AWS Management Console, auto scaling is enabled by default for each replica table, with default auto scaling settings for managing read capacity units and write capacity units.

Changes to auto scaling settings for a replica table or secondary index made through the DynamoDB console or using the `UpdateGlobalTableSettings` call are applied to all of the replica tables and matching secondary indexes in the global table automatically. These changes overwrite any existing auto scaling settings. This ensures that provisioned write capacity settings are consistent across the replica tables and secondary indexes in your global table. If you use the `UpdateTable`, `RegisterScalableTarget`, or `PutScalingPolicy` calls, you should apply the change to each replica table and matching secondary index individually.

Note

If auto scaling doesn't satisfy your application's capacity changes (unpredictable workload), or if you don't want to configure its settings (target settings for minimum, maximum, or

utilization threshold), you can use on-demand mode to manage capacity for your global tables. For more information, see On-demand mode.
If you enable on-demand mode on a global table, your consumption of replicated write request units (rWCUs) will be consistent with how rWCUs are provisioned. For example, if you perform 10 writes to a local table that is replicated in two additional Regions, you will consume 60 write request units (10 + 10 + 10 = 30; 30 x 2 = 60). The consumed 60 write request units include the extra write consumed by global tables Version 2017.11.29 (Legacy) to update the `aws:rep:deleting`, `aws:rep:updatetime`, and `aws:rep:updateregion` attributes.

Managing capacity manually

If you decide not to use DynamoDB auto scaling, you must manually set the read capacity and write capacity settings on each replica table and secondary index.

The provisioned replicated write capacity units (rWCUs) on every replica table should be set to the total number of rWCUs needed for application writes across all Regions multiplied by two. This accommodates application writes that occur in the local Region and replicated application writes coming from other Regions. For example, suppose that you expect 5 writes per second to your replica table in Ohio and 5 writes per second to your replica table in N. Virginia. In this case, you should provision 20 rWCUs to each replica table (5 + 5 = 10; 10 x 2 = 20).

To update write capacity settings for your global table, we strongly recommend using the DynamoDB console or the `UpdateGlobalTableSettings` API operation. `UpdateGlobalTableSettings` applies changes to write capacity settings to all replica tables and matching secondary indexes in a global table automatically. If you use the `UpdateTable`, `RegisterScalableTarget`, or `PutScalingPolicy` operations, you should apply the change to each replica table and matching secondary index individually. For more information, see Amazon DynamoDB API Reference.

Note

To update the settings (`UpdateGlobalTableSettings`) for a global table in DynamoDB, you must have the `dynamodb:UpdateGlobalTable`, `dynamodb:DescribeLimits`, `application-autoscaling:DeleteScalingPolicy`, and `application-autoscaling:DeregisterScalableTarget` permissions. For more information, see Using IAM with global tables.

Creating a global table (Version 2017.11.29)

> **Important**
>
> This documentation is for version 2017.11.29 (Legacy) of global tables, which should be avoided for new global tables. Customers should use Global Tables version 2019.11.21 (Current) when possible, as it provides greater flexibility, higher efficiency and consumes less write capacity than 2017.11.29 (Legacy).
> To determine which version you are using, see Determining the version of a global table.
> To update existing global tables from version 2017.11.29 (Legacy) to version 2019.11.21 (Current), see DynamoDB global tables versions.

This section describes how to create a global table using the Amazon DynamoDB console or the AWS Command Line Interface (AWS CLI).

Topics

- Creating a global table (console)
- Creating a global table (AWS CLI)

Creating a global table (console)

Follow these steps to create a global table using the console. The following example creates a global table with replica tables in United States and Europe.

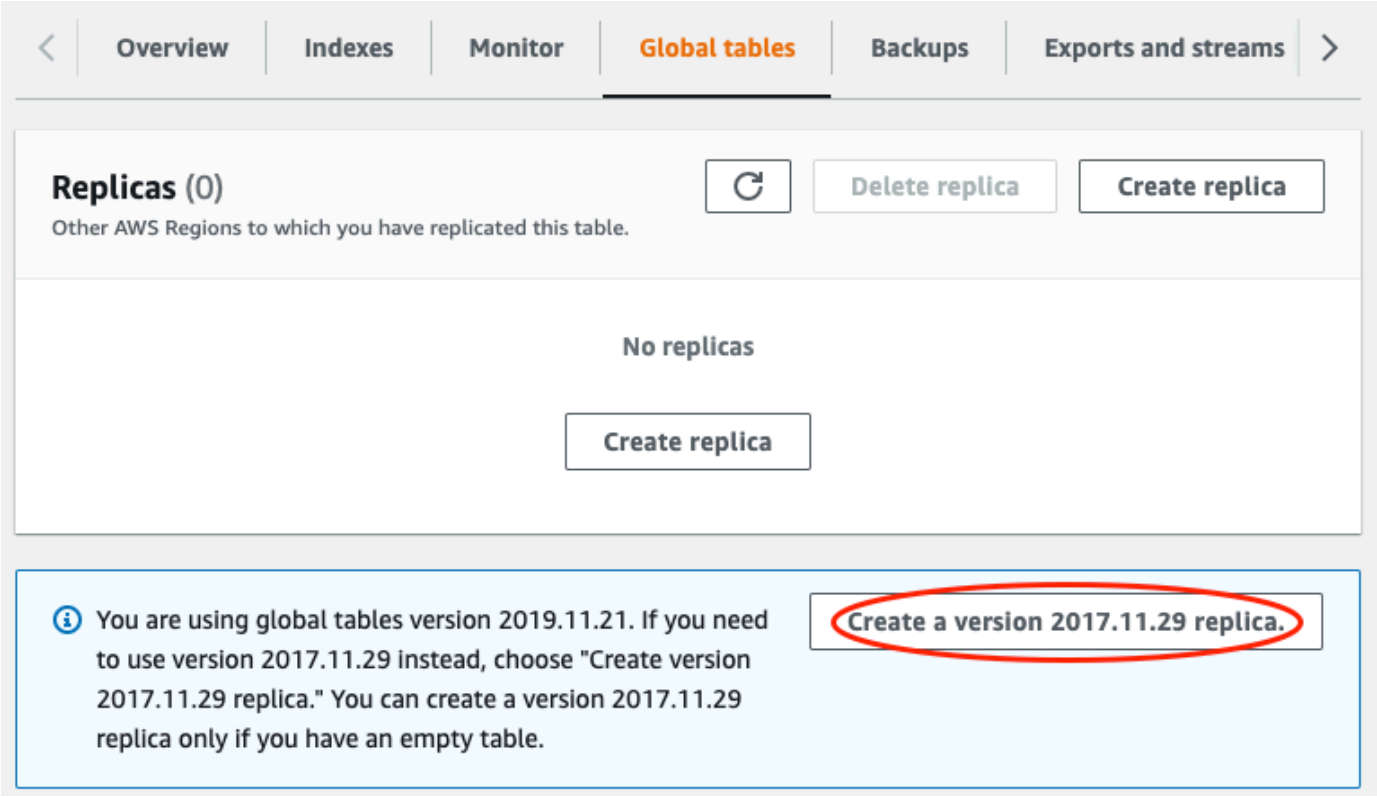
1. Open the DynamoDB console at https://console.aws.amazon.com/dynamodb/home. For this example, choose the us-east-2 (US East Ohio) Region.
2. In the navigation pane on the left side of the console, choose **Tables**.
3. Choose **Create Table**.

   For **Table name**, enter **`Music`**.

   For **Primary key** enter **`Artist`**. Choose **Add sort key**, and enter **`SongTitle`**. (**`Artist`** and **`SongTitle`** should both be strings.)

   To create the table, choose **Create**. This table serves as the first replica table in a new global table. It is the prototype for other replica tables that you add later.

4. Choose the **Global Tables** tab, and then choose **Create a Version 2017.11.29 (Legacy) replica**.



5. From the **Available replication Regions** dropdown, choose **US West (Oregon)**.

   The console checks to ensure that a table with the same name doesn't exist in the selected Region. If a table with the same name does exist, you must delete the existing table before you can create a new replica table in that Region.

6. Choose **Create Replica**. This starts the table creation process in US West (Oregon);.

   The **Global Table** tab for the selected table (and for any other replica tables) shows that the table has been replicated in multiple Regions.

7. Now add another Region so that your global table is replicated and synchronized across the United States and Europe. To do this, repeat step 5, but this time, specify **Europe (Frankfurt)** instead of **US West (Oregon)**.

8. You should still be using the AWS Management Console in the US East (Ohio) Region. Select **Items** in the left navigation menu, select the **Music** table, then choose **Create Item**.

   a. For **Artist**, enter **`item_1`**.

   b. For **SongTitle**, enter **`Song Value 1`**.

   c. To write the item, choose **Create item**.

9. After a short time, the item is replicated across all three Regions of your global table. To verify this, in the console, on the Region selector in the upper-right corner, choose **Europe (Frankfurt)**. The `Music` table in Europe (Frankfurt) should contain the new item.
10. Repeat step 9 and choose **US West (Oregon)** to verify replication in that region.

Creating a global table (AWS CLI)

Follow these steps to create a global table `Music` using the AWS CLI. The following example creates a global table, with replica tables in the United States and in Europe.

1. Create a new table (`Music`) in US East (Ohio), with DynamoDB Streams enabled (`NEW_AND_OLD_IMAGES`).

```
aws dynamodb create-table \
    --table-name Music \
    --attribute-definitions \
        AttributeName=Artist,AttributeType=S \
        AttributeName=SongTitle,AttributeType=S \
    --key-schema \
        AttributeName=Artist,KeyType=HASH \
        AttributeName=SongTitle,KeyType=RANGE \
    --provisioned-throughput \
        ReadCapacityUnits=10,WriteCapacityUnits=5 \
    --stream-specification StreamEnabled=true,StreamViewType=NEW_AND_OLD_IMAGES \
    --region us-east-2
```

2. Create an identical `Music` table in US East (N. Virginia).

```
aws dynamodb create-table \
    --table-name Music \
    --attribute-definitions \
        AttributeName=Artist,AttributeType=S \
        AttributeName=SongTitle,AttributeType=S \
    --key-schema \
        AttributeName=Artist,KeyType=HASH \
        AttributeName=SongTitle,KeyType=RANGE \
    --provisioned-throughput \
        ReadCapacityUnits=10,WriteCapacityUnits=5 \
    --stream-specification StreamEnabled=true,StreamViewType=NEW_AND_OLD_IMAGES \
    --region us-east-1
```

3. Create a global table (`Music`) consisting of replica tables in the `us-east-2` and `us-east-1` Regions.

```
aws dynamodb create-global-table \
    --global-table-name Music \
    --replication-group RegionName=us-east-2 RegionName=us-east-1 \
    --region us-east-2
```

> **Note**
>
> The global table name (`Music`) must match the name of each of the replica tables (`Music`). For more information, see Best practices for global tables.

4. Create another table in Europe (Ireland), with the same settings as those you created in step 1 and step 2.

```
aws dynamodb create-table \
    --table-name Music \
    --attribute-definitions \
        AttributeName=Artist,AttributeType=S \
        AttributeName=SongTitle,AttributeType=S \
    --key-schema \
        AttributeName=Artist,KeyType=HASH \
        AttributeName=SongTitle,KeyType=RANGE \
    --provisioned-throughput \
        ReadCapacityUnits=10,WriteCapacityUnits=5 \
    --stream-specification StreamEnabled=true,StreamViewType=NEW_AND_OLD_IMAGES \
    --region eu-west-1
```

After doing this step, add the new table to the `Music` global table.

```
aws dynamodb update-global-table \
    --global-table-name Music \
    --replica-updates 'Create={RegionName=eu-west-1}' \
    --region us-east-2
```

5. To verify that replication is working, add a new item to the `Music` table in US East (Ohio).

```
aws dynamodb put-item \
    --table-name Music \
```

```
    --item '{"Artist": {"S":"item_1"},"SongTitle": {"S":"Song Value 1"}}' \
    --region us-east-2
```

6. Wait for a few seconds, and then check to see whether the item has been successfully replicated to US East (N. Virginia) and Europe (Ireland).

```
aws dynamodb get-item \
    --table-name Music \
    --key '{"Artist": {"S":"item_1"},"SongTitle": {"S":"Song Value 1"}}' \
    --region us-east-1
```

```
aws dynamodb get-item \
    --table-name Music \
    --key '{"Artist": {"S":"item_1"},"SongTitle": {"S":"Song Value 1"}}' \
    --region eu-west-1
```

Monitoring global tables

> **Important**
>
> This documentation is for version 2017.11.29 (Legacy) of global tables, which should be avoided for new global tables. Customers should use Global Tables version 2019.11.21 (Current) when possible, as it provides greater flexibility, higher efficiency and consumes less write capacity than 2017.11.29 (Legacy).
> To determine which version you are using, see Determining the version of a global table.
> To update existing global tables from version 2017.11.29 (Legacy) to version 2019.11.21 (Current), see DynamoDB global tables versions.

You can use Amazon CloudWatch to monitor the behavior and performance of a global table. Amazon DynamoDB publishes `ReplicationLatency` and `PendingReplicationCount` metrics for each replica in the global table.

- **`ReplicationLatency`**—The elapsed time between when an updated item appears in the DynamoDB stream for one replica table, and when that item appears in another replica in the global table. `ReplicationLatency` is expressed in milliseconds and is emitted for every source- and destination-Region pair.

During normal operation, `ReplicationLatency` should be fairly constant. An elevated value for `ReplicationLatency` could indicate that updates from one replica are not propagating to other replica tables in a timely manner. Over time, this could result in other replica tables *falling behind* because they no longer receive updates consistently. In this case, you should verify that the read capacity units (RCUs) and write capacity units (WCUs) are identical for each of the replica tables. In addition, when choosing WCU settings, follow the recommendations in Global tables version.

`ReplicationLatency` can increase if an AWS Region becomes degraded and you have a replica table in that Region. In this case, you can temporarily redirect your application's read and write activity to a different AWS Region.

- **`PendingReplicationCount`**—The number of item updates that are written to one replica table, but that have not yet been written to another replica in the global table. `PendingReplicationCount` is expressed in number of items and is emitted for every source- and destination-Region pair.

  During normal operation, `PendingReplicationCount` should be very low. If `PendingReplicationCount` increases for extended periods, investigate whether your replica tables' provisioned write capacity settings are sufficient for your current workload.

  `PendingReplicationCount` can increase if an AWS Region becomes degraded and you have a replica table in that Region. In this case, you can temporarily redirect your application's read and write activity to a different AWS Region.

For more information, see DynamoDB Metrics and dimensions.

Using IAM with global tables

> **Important**
>
> This documentation is for version 2017.11.29 (Legacy) of global tables, which should be avoided for new global tables. Customers should use Global Tables version 2019.11.21 (Current) when possible, as it provides greater flexibility, higher efficiency and consumes less write capacity than 2017.11.29 (Legacy).

To determine which version you are using, see Determining the version of a global table. To update existing global tables from version 2017.11.29 (Legacy) to version 2019.11.21 (Current), see DynamoDB global tables versions.

When you create a global table for the first time, Amazon DynamoDB automatically creates an AWS Identity and Access Management (IAM) service-linked role for you. This role is named , and it allows DynamoDB to manage cross-Region replication for global tables on your behalf. Don't delete this service-linked role. If you do, then all of your global tables will no longer function.

For more information about service-linked roles, see Using service-linked roles in the *IAM User Guide*.

To create and maintain global tables in DynamoDB, you must have the `dynamodb:CreateGlobalTable` permission to access each of the following:

- The replica table that you want to add.
- Each existing replica that's already part of the global table.
- The global table itself.

To update the settings (`UpdateGlobalTableSettings`) for a global table in DynamoDB, you must have the `dynamodb:UpdateGlobalTable`, `dynamodb:DescribeLimits`, `application-autoscaling:DeleteScalingPolicy`, and `application-autoscaling:DeregisterScalableTarget` permissions.

The `application-autoscaling:DeleteScalingPolicy` and `application-autoscaling:DeregisterScalableTarget` permissions are required when updating an existing scaling policy. This is so that the global tables service can remove the old scaling policy before attaching the new policy to the table or secondary index.

If you use an IAM policy to manage access to one replica table, you should apply an identical policy to all other replicas within that global table. This practice helps you maintain a consistent permissions model across all of the replica tables.

By using identical IAM policies on all replicas in a global table, you can also avoid granting unintended read and write access to your global table data. For example, consider a user who has access to only one replica in a global table. If that user can write to this replica, DynamoDB propagates the write to all of the other replica tables. In effect, the user can (indirectly) write to

all of the other replicas in the global table. This scenario can be avoided by using consistent IAM policies on all of the replica tables.

Example: Allow the CreateGlobalTable action

Before you can add a replica to a global table, you must have the `dynamodb:CreateGlobalTable` permission for the global table and for each of its replica tables.

The following IAM policy grants permissions to allow the `CreateGlobalTable` action on all tables.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": ["dynamodb:CreateGlobalTable"],
            "Resource": "*"
        }
    ]
}
```

Example: Allow the UpdateGlobalTable, DescribeLimits, application-autoscaling:DeleteScalingPolicy, and application-autoscaling:DeregisterScalableTarget actions

To update the settings (`UpdateGlobalTableSettings`) for a global table in DynamoDB, you must have the `dynamodb:UpdateGlobalTable`, `dynamodb:DescribeLimits`, `application-autoscaling:DeleteScalingPolicy`, and `application-autoscaling:DeregisterScalableTarget` permissions.

The following IAM policy grants permissions to allow the `UpdateGlobalTableSettings` action on all tables.

JSON

```
{
```

```
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "dynamodb:UpdateGlobalTable",
                "dynamodb:DescribeLimits",
                "application-autoscaling:DeleteScalingPolicy",
                "application-autoscaling:DeregisterScalableTarget"
            ],
            "Resource": "*"
        }
    ]
}
```

Example: Allow the CreateGlobalTable action for a specific global table name with replicas allowed in certain regions only

The following IAM policy grants permissions to allow the `CreateGlobalTable` action to create a global table named `Customers` with replicas in two Regions.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "dynamodb:CreateGlobalTable",
            "Resource": [
                "arn:aws:dynamodb::123456789012:global-table/Customers",
                "arn:aws:dynamodb:us-east-1:123456789012:table/Customers",
                "arn:aws:dynamodb:us-west-1:123456789012:table/Customers"
            ]
        }
    ]
}
```

Previous low-level DynamoDB API version (2011-12-05)

This section documents the operations available in the previous DynamoDB low-level API version (2011-12-05). This version of the low-level API is maintained for backward compatibility with existing applications.

New applications should use the current API version (2012-08-10). For more information, see DynamoDB API reference.

> **Note**
>
> We recommend that you migrate your applications to the latest API version (2012-08-10), since new DynamoDB features will not be backported to the previous API version.

Topics

- BatchGetItem
- BatchWriteItem
- CreateTable
- DeleteItem
- DeleteTable
- DescribeTables
- GetItem
- ListTables
- PutItem
- Query
- Scan
- UpdateItem
- UpdateTable

BatchGetItem

Important

This section refers to API version 2011-12-05, which is deprecated and should not be used for new applications.

For documentation on the current low-level API, see the Amazon DynamoDB API Reference.

Description

The `BatchGetItem` operation returns the attributes for multiple items from multiple tables using their primary keys. The maximum number of items that can be retrieved for a single operation is 100. Also, the number of items retrieved is constrained by a 1 MB size limit. If the response size limit is exceeded or a partial result is returned because the table's provisioned throughput is exceeded, or because of an internal processing failure, DynamoDB returns an `UnprocessedKeys` value so you can retry the operation starting with the next item to get. DynamoDB automatically adjusts the number of items returned per page to enforce this limit. For example, even if you ask to retrieve 100 items, but each individual item is 50 KB in size, the system returns 20 items and an appropriate `UnprocessedKeys` value so you can get the next page of results. If desired, your application can include its own logic to assemble the pages of results into one set.

If no items could be processed because of insufficient provisioned throughput on each of the tables involved in the request, DynamoDB returns a `ProvisionedThroughputExceededException` error.

Note

By default, `BatchGetItem` performs eventually consistent reads on every table in the request. You can set the `ConsistentRead` parameter to `true`, on a per-table basis, if you want consistent reads instead.

`BatchGetItem` fetches items in parallel to minimize response latencies.

When designing your application, keep in mind that DynamoDB does not guarantee how attributes are ordered in the returned response. Include the primary key values in the `AttributesToGet` for the items in your request to help parse the response by item.

If the requested items do not exist, nothing is returned in the response for those items. Requests for non-existent items consume the minimum read capacity units according to the type of read. For more information, see DynamoDB item sizes and formats.

Requests

Syntax

```
// This header is abbreviated. For a sample of a complete header, see DynamoDB low-level API.
POST / HTTP/1.1
x-amz-target: DynamoDB_20111205.BatchGetItem
content-type: application/x-amz-json-1.0

{"RequestItems":
    {"Table1":
        {"Keys":
            [{"HashKeyElement": {"S":"KeyValue1"}, "RangeKeyElement":
{"N":"KeyValue2"}},
            {"HashKeyElement": {"S":"KeyValue3"}, "RangeKeyElement":{"N":"KeyValue4"}},
            {"HashKeyElement": {"S":"KeyValue5"}, "RangeKeyElement":
{"N":"KeyValue6"}}],
        "AttributesToGet":["AttributeName1", "AttributeName2", "AttributeName3"]},
    "Table2":
        {"Keys":
            [{"HashKeyElement": {"S":"KeyValue4"}},
            {"HashKeyElement": {"S":"KeyValue5"}}],
        "AttributesToGet": ["AttributeName4", "AttributeName5", "AttributeName6"]
        }
    }
}
```

Name	Description	Required
`RequestItems`	A container of the table name and corresponding items to get by primary key. While requesting items, each table	Yes

Name	Description	Required
	name can be invoked only once per operation. Type: String Default: None	
`Table`	The name of the table containing the items to get. The entry is simply a string specifying an existing table with no label. Type: String Default: None	Yes
`Table:Keys`	The primary key values that define the items in the specified table. For more information about primary keys, see Primary key. Type: Keys	Yes
`Table:AttributesToGet`	Array of Attribute names within the specified table. If attribute names are not specified then all attribute s will be returned. If some attributes are not found, they will not appear in the result. Type: Array	No

Name	Description	Required
Table:ConsistentRead	If set to `true`, then a consistent read is issued, otherwise eventually consistent is used. Type: Boolean	No

Responses

Syntax

```
HTTP/1.1 200
x-amzn-RequestId: 8966d095-71e9-11e0-a498-71d736f27375
content-type: application/x-amz-json-1.0
content-length: 855

{"Responses":
    {"Table1":
        {"Items":
        [{"AttributeName1": {"S":"AttributeValue"},
        "AttributeName2": {"N":"AttributeValue"},
        "AttributeName3": {"SS":["AttributeValue", "AttributeValue", "AttributeValue"]}
        },
        {"AttributeName1": {"S": "AttributeValue"},
        "AttributeName2": {"S": "AttributeValue"},
        "AttributeName3": {"NS": ["AttributeValue", "AttributeValue",
 "AttributeValue"]}
        }],
    "ConsumedCapacityUnits":1},
    "Table2":
        {"Items":
        [{"AttributeName1": {"S":"AttributeValue"},
        "AttributeName2": {"N":"AttributeValue"},
        "AttributeName3": {"SS":["AttributeValue", "AttributeValue", "AttributeValue"]}
        },
        {"AttributeName1": {"S": "AttributeValue"},
        "AttributeName2": {"S": "AttributeValue"},
        "AttributeName3": {"NS": ["AttributeValue", "AttributeValue","AttributeValue"]}
        }],
```

```
    "ConsumedCapacityUnits":1}
    },
    "UnprocessedKeys":
        {"Table3":
        {"Keys":
            [{"HashKeyElement": {"S":"KeyValue1"}, "RangeKeyElement":
{"N":"KeyValue2"}},
            {"HashKeyElement": {"S":"KeyValue3"}, "RangeKeyElement":{"N":"KeyValue4"}},
            {"HashKeyElement": {"S":"KeyValue5"}, "RangeKeyElement":
{"N":"KeyValue6"}}],
        "AttributesToGet":["AttributeName1", "AttributeName2", "AttributeName3"]}
        }
}
```

Name	Description
Responses	Table names and the respective item attributes from the tables. Type: Map
Table	The name of the table containing the items. The entry is simply a string specifying the table with no label. Type: String
Items	Container for the attribute names and values meeting the operation parameters. Type: Map of attribute names to and their data types and values.
ConsumedCapacityUnits	The number of read capacity units consumed, for each table. This value shows the number applied toward your provisioned throughput. Requests for non-existent items consume the minimum read capacity units, depending on the type of read. For more information see DynamoDB provisioned capacity mode.

Name	Description
	Type: Number
UnprocessedKeys	Contains an array of tables and their respective keys that were not processed with the current response, possibly due to reaching a limit on the response size. The UnprocessedKeys value is in the same form as a RequestItems parameter (so the value can be provided directly to a subsequent BatchGetItem operation). For more information, see the above RequestItems parameter. Type: Array
UnprocessedKeys : Table: Keys	The primary key attribute values that define the items and the attributes associated with the items. For more information about primary keys, see Primary key . Type: Array of attribute name-value pairs.
UnprocessedKeys : Table: AttributesToGet	Attribute names within the specified table. If attribute names are not specified then all attributes will be returned. If some attributes are not found, they will not appear in the result. Type: Array of attribute names.
UnprocessedKeys : Table: ConsistentRead	If set to true, then a consistent read is used for the specified table, otherwise an eventually consistent read is used. Type: Boolean.

Special errors

Error	Description
`ProvisionedThroughputExceededException`	Your maximum allowed provisioned throughput has been exceeded.

Examples

The following examples show an HTTP POST request and response using the BatchGetItem operation. For examples using the AWS SDK, see Working with items and attributes in DynamoDB.

Sample request

The following sample requests attributes from two different tables.

```
// This header is abbreviated.
// For a sample of a complete header, see DynamoDB low-level API.
POST / HTTP/1.1
x-amz-target: DynamoDB_20111205.BatchGetItem
content-type: application/x-amz-json-1.0
content-length: 409

{"RequestItems":
    {"comp1":
        {"Keys":
            [{"HashKeyElement":{"S":"Casey"},"RangeKeyElement":{"N":"1319509152"}},
            {"HashKeyElement":{"S":"Dave"},"RangeKeyElement":{"N":"1319509155"}},
            {"HashKeyElement":{"S":"Riley"},"RangeKeyElement":{"N":"1319509158"}}],
        "AttributesToGet":["user","status"]},
    "comp2":
        {"Keys":
            [{"HashKeyElement":{"S":"Julie"}},{"HashKeyElement":{"S":"Mingus"}}],
        "AttributesToGet":["user","friends"]}
    }
}
```

Sample response

The following sample is the response.

```
HTTP/1.1 200 OK
x-amzn-RequestId: GTPQVRM4VJS792J1UFJTKUBVV4KQNSO5AEMVJF66Q9ASUAAJG
content-type: application/x-amz-json-1.0
content-length: 373
Date: Fri, 02 Sep 2011 23:07:39 GMT

{"Responses":
    {"comp1":
        {"Items":
            [{"status":{"S":"online"},"user":{"S":"Casey"}},
            {"status":{"S":"working"},"user":{"S":"Riley"}},
            {"status":{"S":"running"},"user":{"S":"Dave"}}],
        "ConsumedCapacityUnits":1.5},
    "comp2":
        {"Items":
            [{"friends":{"SS":["Elisabeth", "Peter"]},"user":{"S":"Mingus"}},
            {"friends":{"SS":["Dave", "Peter"]},"user":{"S":"Julie"}}],
        "ConsumedCapacityUnits":1}
    },
    "UnprocessedKeys":{}
}
```

BatchWriteItem

> **Important**
>
> ***This section refers to API version 2011-12-05, which is deprecated and should not be used for new applications.***
>
> **For documentation on the current low-level API, see the Amazon DynamoDB API Reference.**

Description

This operation enables you to put or delete several items across multiple tables in a single call.

To upload one item, you can use `PutItem`, and to delete one item, you can use `DeleteItem`. However, when you want to upload or delete large amounts of data, such as uploading large amounts of data from Amazon EMR (Amazon EMR) or migrating data from another database in to DynamoDB, `BatchWriteItem` offers an efficient alternative.

If you use languages such as Java, you can use threads to upload items in parallel. This adds complexity in your application to handle the threads. Other languages don't support threading. For example, if you are using PHP, you must upload or delete items one at a time. In both situations, `BatchWriteItem` provides an alternative where the specified put and delete operations are processed in parallel, giving you the power of the thread pool approach without having to introduce complexity in your application.

Note that each individual put and delete specified in a `BatchWriteItem` operation costs the same in terms of consumed capacity units. However, because `BatchWriteItem` performs the specified operations in parallel, you get lower latency. Delete operations on non-existent items consume 1 write capacity unit. For more information about consumed capacity units, see Working with tables and data in DynamoDB.

When using `BatchWriteItem`, note the following limitations:

- **Maximum operations in a single request—**You can specify a total of up to 25 put or delete operations; however, the total request size cannot exceed 1 MB (the HTTP payload).
- You can use the `BatchWriteItem` operation only to put and delete items. You cannot use it to update existing items.
- **Not an atomic operation—**Individual operations specified in a `BatchWriteItem` are atomic; however `BatchWriteItem` as a whole is a "best-effort" operation and not an atomic operation. That is, in a `BatchWriteItem` request, some operations might succeed and others might fail. The failed operations are returned in an `UnprocessedItems` field in the response. Some of these failures might be because you exceeded the provisioned throughput configured for the table or a transient failure such as a network error. You can investigate and optionally resend the requests. Typically, you call `BatchWriteItem` in a loop and in each iteration check for unprocessed items, and submit a new `BatchWriteItem` request with those unprocessed items.
- **Does not return any items—**The `BatchWriteItem` is designed for uploading large amounts of data efficiently. It does not provide some of the sophistication offered by `PutItem` and `DeleteItem`. For example, `DeleteItem` supports the `ReturnValues` field in your request body to request the deleted item in the response. The `BatchWriteItem` operation does not return any items in the response.
- Unlike `PutItem` and `DeleteItem`, `BatchWriteItem` does not allow you to specify conditions on individual write requests in the operation.
- Attribute values must not be null; string and binary type attributes must have lengths greater than zero; and set type attributes must not be empty. Requests that have empty values will be rejected with a `ValidationException`.

DynamoDB rejects the entire batch write operation if any one of the following is true:

- If one or more tables specified in the `BatchWriteItem` request does not exist.
- If primary key attributes specified on an item in the request does not match the corresponding table's primary key schema.
- If you try to perform multiple operations on the same item in the same `BatchWriteItem` request. For example, you cannot put and delete the same item in the same `BatchWriteItem` request.
- If the total request size exceeds the 1 MB request size (the HTTP payload) limit.
- If any individual item in a batch exceeds the 64 KB item size limit.

Requests

Syntax

```
// This header is abbreviated. For a sample of a complete header, see DynamoDB low-level API.
POST / HTTP/1.1
x-amz-target: DynamoDB_20111205.BatchGetItem
content-type: application/x-amz-json-1.0

{
  "RequestItems" :  RequestItems
}

RequestItems
{
     "TableName1" :  [ Request, Request, ... ],
     "TableName2" :  [ Request, Request, ... ],
     ...
}

Request ::=
  PutRequest | DeleteRequest

PutRequest ::=
{
  "PutRequest" : {
     "Item" : {
        "Attribute-Name1" : Attribute-Value,
```

```
        "Attribute-Name2" : Attribute-Value,
         ...
     }
  }
}

DeleteRequest ::=
{
   "DeleteRequest" : {
      "Key" : PrimaryKey-Value
   }
}


PrimaryKey-Value ::= HashTypePK | HashAndRangeTypePK

HashTypePK ::=
{
   "HashKeyElement" : Attribute-Value
}

HashAndRangeTypePK
{
   "HashKeyElement" : Attribute-Value,
   "RangeKeyElement" : Attribute-Value,
}

Attribute-Value ::= String | Numeric| Binary | StringSet | NumericSet | BinarySet

Numeric ::=
{
   "N": "Number"
}

String ::=
{
   "S": "String"
}

Binary ::=
{
    "B": "Base64 encoded binary data"
}
```

```
StringSet ::=
{
   "SS": [ "String1", "String2", ... ]
}

NumberSet ::=
{
   "NS": [ "Number1", "Number2", ... ]
}

BinarySet ::=
{
   "BS": [ "Binary1", "Binary2", ... ]
}
```

In the request body, the `RequestItems` JSON object describes the operations that you want to perform. The operations are grouped by tables. You can use `BatchWriteItem` to update or delete several items across multiple tables. For each specific write request, you must identify the type of request (`PutItem`, `DeleteItem`) followed by detail information about the operation.

- For a `PutRequest`, you provide the item, that is, a list of attributes and their values.
- For a `DeleteRequest`, you provide the primary key name and value.

Responses

Syntax

The following is the syntax of the JSON body returned in the response.

```
{
  "Responses" :         ConsumedCapacityUnitsByTable
  "UnprocessedItems" :  RequestItems
}

ConsumedCapacityUnitsByTable
{
    "TableName1" : { "ConsumedCapacityUnits", : NumericValue },
    "TableName2" : { "ConsumedCapacityUnits", : NumericValue },
     ...
}

RequestItems
```

```
This syntax is identical to the one described in the JSON syntax in the request.
```

Special errors

No errors specific to this operation.

Examples

The following example shows an HTTP POST request and the response of a `BatchWriteItem` operation. The request specifies the following operations on the Reply and the Thread tables:

- Put an item and delete an item from the Reply table
- Put an item into the Thread table

For examples using the AWS SDK, see Working with items and attributes in DynamoDB.

Sample request

```
// This header is abbreviated. For a sample of a complete header, see DynamoDB low-
level API.
POST / HTTP/1.1
x-amz-target: DynamoDB_20111205.BatchGetItem
content-type: application/x-amz-json-1.0

{
  "RequestItems":{
    "Reply":[
      {
        "PutRequest":{
          "Item":{
            "ReplyDateTime":{
              "S":"2012-04-03T11:04:47.034Z"
            },
            "Id":{
              "S":"DynamoDB#DynamoDB Thread 5"
            }
          }
        }
      },
      {
        "DeleteRequest":{
          "Key":{
```

```
            "HashKeyElement":{
              "S":"DynamoDB#DynamoDB Thread 4"
            },
            "RangeKeyElement":{
              "S":"oops - accidental row"
            }
          }
        }
      }
    ],
    "Thread":[
      {
        "PutRequest":{
          "Item":{
            "ForumName":{
              "S":"DynamoDB"
            },
            "Subject":{
              "S":"DynamoDB Thread 5"
            }
          }
        }
      }
    ]
  }
}
```

Sample response

The following example response shows a put operation on both the Thread and Reply tables succeeded and a delete operation on the Reply table failed (for reasons such as throttling that is caused when you exceed the provisioned throughput on the table). Note the following in the JSON response:

- The `Responses` object shows one capacity unit was consumed on both the `Thread` and `Reply` tables as a result of the successful put operation on each of these tables.
- The `UnprocessedItems` object shows the unsuccessful delete operation on the `Reply` table. You can then issue a new `BatchWriteItem` call to address these unprocessed requests.

```
HTTP/1.1 200 OK
x-amzn-RequestId: G8M9ANLOE5QA26AEUHJKJE0ASBVV4KQNSO5AEMVJF66Q9ASUAAJG
```

```
Content-Type: application/x-amz-json-1.0
Content-Length: 536
Date: Thu, 05 Apr 2012 18:22:09 GMT

{
   "Responses":{
      "Thread":{
         "ConsumedCapacityUnits":1.0
      },
      "Reply":{
         "ConsumedCapacityUnits":1.0
      }
   },
   "UnprocessedItems":{
      "Reply":[
         {
            "DeleteRequest":{
               "Key":{
                  "HashKeyElement":{
                     "S":"DynamoDB#DynamoDB Thread 4"
                  },
                  "RangeKeyElement":{
                     "S":"oops - accidental row"
                  }
               }
            }
         }
      ]
   }
}
```

CreateTable

Important

This section refers to API version 2011-12-05, which is deprecated and should not be used for new applications.

For documentation on the current low-level API, see the Amazon DynamoDB API Reference.

Description

The `CreateTable` operation adds a new table to your account.

The table name must be unique among those associated with the AWS Account issuing the request, and the AWS region that receives the request (such as dynamodb.us-west-2.amazonaws.com). Each DynamoDB endpoint is entirely independent. For example, if you have two tables called "MyTable," one in dynamodb.us-west-2.amazonaws.com and one in dynamodb.us-west-1.amazonaws.com, they are completely independent and do not share any data.

The `CreateTable` operation triggers an asynchronous workflow to begin creating the table. DynamoDB immediately returns the state of the table (`CREATING`) until the table is in the `ACTIVE` state. Once the table is in the `ACTIVE` state, you can perform data plane operations.

Use the DescribeTables operation to check the status of the table.

Requests

Syntax

```
// This header is abbreviated.
// For a sample of a complete header, see DynamoDB low-level API.
POST / HTTP/1.1
x-amz-target: DynamoDB_20111205.CreateTable
content-type: application/x-amz-json-1.0

{"TableName":"Table1",
    "KeySchema":
        {"HashKeyElement":{"AttributeName":"AttributeName1","AttributeType":"S"},
        "RangeKeyElement":{"AttributeName":"AttributeName2","AttributeType":"N"}},
    "ProvisionedThroughput":{"ReadCapacityUnits":5,"WriteCapacityUnits":10}
}
```

Name	Description	Required
`TableName`	The name of the table to create. Allowed characters are a-z, A-Z, 0-9, '_' (underscore),	Yes

Name	Description	Required
	'-' (dash), and '.' (dot). Names can be between 3 and 255 characters long. Type: String	
KeySchema	The primary key (simple or composite) structure for the table. A name-value pair for the HashKeyElement is required, and a name-value pair for the RangeKeyElement is optional (only required for composite primary keys). For more information about primary keys, see Primary key. Primary key element names can be between 1 and 255 characters long with no character restrictions. Possible values for the AttributeType are "S" (string), "N" (numeric), or "B" (binary). Type: Map of HashKeyElement, or HashKeyElement and RangeKeyElement for a composite primary key.	Yes

Name	Description	Required
`ProvisionedThroughput`	New throughput for the specified table, consisting of values for `ReadCapacityUnits` and `WriteCapacityUnits`. For details, see DynamoDB provisioned capacity mode. **Note** For current maximum/minimum values, see Quotas in Amazon DynamoDB. Type: Array	Yes

Name	Description	Required
`ProvisionedThroughput : ReadCapacityUnits`	Sets the minimum number of consistent `ReadCapacityUnits` consumed per second for the specified table before DynamoDB balances the load with other operations. Eventually consistent read operations require less effort than a consistent read operation, so a setting of 50 consistent `ReadCapacityUnits` per second provides 100 eventually consistent `ReadCapacityUnits` per second. Type: Number	Yes
`ProvisionedThroughput : WriteCapacityUnits`	Sets the minimum number of `WriteCapacityUnits` consumed per second for the specified table before DynamoDB balances the load with other operations. Type: Number	Yes

Responses

Syntax

```
HTTP/1.1 200 OK
x-amzn-RequestId: CSOC7TJPLR0OOKIRLGOHVAICUFVV4KQNSO5AEMVJF66Q9ASUAAJG
```

```
content-type: application/x-amz-json-1.0
content-length: 311
Date: Tue, 12 Jul 2011 21:31:03 GMT

{"TableDescription":
    {"CreationDateTime":1.310506263362E9,
    "KeySchema":
        {"HashKeyElement":{"AttributeName":"AttributeName1","AttributeType":"S"},
        "RangeKeyElement":{"AttributeName":"AttributeName2","AttributeType":"N"}},
    "ProvisionedThroughput":{"ReadCapacityUnits":5,"WriteCapacityUnits":10},
    "TableName":"Table1",
    "TableStatus":"CREATING"
    }
}
```

Name	Description
TableDescription	A container for the table properties.
CreationDateTime	Date when the table was created in UNIX epoch time. Type: Number
KeySchema	The primary key (simple or composite) structure for the table. A name-value pair for the HashKeyElement is required, and a name-value pair for the RangeKeyElement is optional (only required for composite primary keys). For more information about primary keys, see Primary key . Type: Map of HashKeyElement , or HashKeyElement and RangeKeyElement for a composite primary key.
ProvisionedThroughput	Throughput for the specified table, consisting of values for ReadCapacityUnits and

Name	Description
	WriteCapacityUnits . See DynamoDB provisioned capacity mode. Type: Array
ProvisionedThroughput :ReadCapacityUnits	The minimum number of ReadCapacityUnits consumed per second before DynamoDB. balances the load with other operations Type: Number
ProvisionedThroughput :WriteCapacityUnits	The minimum number of ReadCapacityUnits consumed per second before WriteCapacityUnits . balances the load with other operations Type: Number
TableName	The name of the created table. Type: String
TableStatus	The current state of the table (CREATING). Once the table is in the ACTIVE state, you can put data in it. Use the DescribeTables API to check the status of the table. Type: String

Special errors

Error	Description
`ResourceInUseException`	Attempt to recreate an already existing table.
`LimitExceededException`	The number of simultaneous table requests (cumulative number of tables in the `CREATING`, `DELETING` or `UPDATING` state) exceeds the maximum allowed. **Note** For current maximum/minimum values, see Quotas in Amazon DynamoDB. .

Examples

The following example creates a table with a composite primary key containing a string and a number. For examples using the AWS SDK, see Working with tables and data in DynamoDB.

Sample request

```
// This header is abbreviated.
// For a sample of a complete header, see DynamoDB low-level API.
POST / HTTP/1.1
x-amz-target: DynamoDB_20111205.CreateTable
content-type: application/x-amz-json-1.0


{"TableName":"comp-table",
    "KeySchema":
        {"HashKeyElement":{"AttributeName":"user","AttributeType":"S"},
        "RangeKeyElement":{"AttributeName":"time","AttributeType":"N"}},
    "ProvisionedThroughput":{"ReadCapacityUnits":5,"WriteCapacityUnits":10}
```

```
}
```

Sample response

```
HTTP/1.1 200 OK
x-amzn-RequestId: CSOC7TJPLR0OOKIRLGOHVAICUFVV4KQNSO5AEMVJF66Q9ASUAAJG
content-type: application/x-amz-json-1.0
content-length: 311
Date: Tue, 12 Jul 2011 21:31:03 GMT

{"TableDescription":
    {"CreationDateTime":1.310506263362E9,
    "KeySchema":
        {"HashKeyElement":{"AttributeName":"user","AttributeType":"S"},
        "RangeKeyElement":{"AttributeName":"time","AttributeType":"N"}},
    "ProvisionedThroughput":{"ReadCapacityUnits":5,"WriteCapacityUnits":10},
    "TableName":"comp-table",
    "TableStatus":"CREATING"
    }
}
```

Related actions

- DescribeTables
- DeleteTable

DeleteItem

> **Important**
>
> ***This section refers to API version 2011-12-05, which is deprecated and should not be used for new applications.***
>
> **For documentation on the current low-level API, see the Amazon DynamoDB API Reference.**

Description

Deletes a single item in a table by primary key. You can perform a conditional delete operation that deletes the item if it exists, or if it has an expected attribute value.

Note

If you specify `DeleteItem` without attributes or values, all the attributes for the item are deleted.
Unless you specify conditions, the `DeleteItem` is an idempotent operation; running it multiple times on the same item or attribute does *not* result in an error response.
Conditional deletes are useful for only deleting items and attributes if specific conditions are met. If the conditions are met, DynamoDB performs the delete. Otherwise, the item is not deleted.
You can perform the expected conditional check on one attribute per operation.

Requests

Syntax

```
// This header is abbreviated.
// For a sample of a complete header, see DynamoDB low-level API.
POST / HTTP/1.1
x-amz-target: DynamoDB_20111205.DeleteItem
content-type: application/x-amz-json-1.0

{"TableName":"Table1",
    "Key":
        {"HashKeyElement":{"S":"AttributeValue1"},"RangeKeyElement":
{"N":"AttributeValue2"}},
    "Expected":{"AttributeName3":{"Value":{"S":"AttributeValue3"}}},
    "ReturnValues":"ALL_OLD"}
}
```

Name	Description	Required
`TableName`	The name of the table containing the item to delete. Type: String	Yes
`Key`	The primary key that defines the item. For more informati	Yes

Name	Description	Required
	on about primary keys, see Primary key. Type: Map of `HashKeyElement` to its value and `RangeKeyElement` to its value.	
`Expected`	Designates an attribute for a conditional delete. The `Expected` parameter allows you to provide an attribute name, and whether or not DynamoDB should check to see if the attribute has a particular value before deleting it. Type: Map of attribute names.	No
`Expected:AttributeName`	The name of the attribute for the conditional put. Type: String	No

Name	Description	Required

`Expected:AttributeName: ExpectedAttributeValue`	Use this parameter to specify whether or not a value already exists for the attribute name-value pair.
	
	The following JSON notation deletes the item if the "Color" attribute doesn't exist for that item:
	
	`"Expected" : {"Color":{"Exists":false}}`
	
	The following JSON notation checks to see if the attribute with name "Color" has an existing value of "Yellow" before deleting the item:
	
	`"Expected" : {"Color":{"Exists":true},{"Value":{"S":"Yellow"}}}`
	
	By default, if you use the `Expected` parameter and provide a `Value`, DynamoDB assumes the attribute exists and has a current value to be replaced. So you don't have to specify `{"Exists":true}`, because it is implied. You can shorten the request to:
	
	`"Expected" :`
		No

Name	Description	Required
	`{"Color":{"Value":{"S":"Yellow"}}}` **Note** If you specify `{"Exists":true}` without an attribute value to check, DynamoDB returns an error.	
`ReturnValues`	Use this parameter if you want to get the attribute name-value pairs before they were deleted. Possible parameter values are `NONE` (default) or `ALL_OLD`. If `ALL_OLD` is specified, the content of the old item is returned. If this parameter is not provided or is `NONE`, nothing is returned. Type: String	No

Responses

Syntax

```
HTTP/1.1 200 OK
x-amzn-RequestId: CSOC7TJPLR0OOKIRLGOHVAICUFVV4KQNSO5AEMVJF66Q9ASUAAJG
content-type: application/x-amz-json-1.0
content-length: 353
Date: Tue, 12 Jul 2011 21:31:03 GMT
```

```
{"Attributes":
    {"AttributeName3":{"SS":["AttributeValue3","AttributeValue4","AttributeValue5"]},
    "AttributeName2":{"S":"AttributeValue2"},
    "AttributeName1":{"N":"AttributeValue1"}
    },
"ConsumedCapacityUnits":1
}
```

Name	Description
`Attributes`	If the `ReturnValues` parameter is provided as `ALL_OLD` in the request, DynamoDB returns an array of attribute name-value pairs (essentially, the deleted item). Otherwise, the response contains an empty set. Type: Array of attribute name-value pairs.
`ConsumedCapacityUnits`	The number of write capacity units consumed by the operation. This value shows the number applied toward your provisioned throughput. Delete requests on non-existent items consume 1 write capacity unit. For more information see DynamoDB provisioned capacity mode. Type: Number

Special errors

Error	Description
`ConditionalCheckFailedException`	Conditional check failed. An expected attribute value was not found.

Examples

Sample request

```
// This header is abbreviated.
// For a sample of a complete header, see DynamoDB low-level API.
POST / HTTP/1.1
x-amz-target: DynamoDB_20111205.DeleteItem
content-type: application/x-amz-json-1.0

{"TableName":"comp-table",
    "Key":
        {"HashKeyElement":{"S":"Mingus"},"RangeKeyElement":{"N":"200"}},
    "Expected":
        {"status":{"Value":{"S":"shopping"}}},
    "ReturnValues":"ALL_OLD"
}
```

Sample response

```
HTTP/1.1 200 OK
x-amzn-RequestId: U9809LI6BBFJA5N2R0TB0P017JVV4KQNSO5AEMVJF66Q9ASUAAJG
content-type: application/x-amz-json-1.0
content-length: 353
Date: Tue, 12 Jul 2011 22:31:23 GMT

{"Attributes":
    {"friends":{"SS":["Dooley","Ben","Daisy"]},
    "status":{"S":"shopping"},
    "time":{"N":"200"},
    "user":{"S":"Mingus"}
    },
"ConsumedCapacityUnits":1
}
```

Related actions

- PutItem

DeleteTable

Important

This section refers to API version 2011-12-05, which is deprecated and should not be used for new applications.

For documentation on the current low-level API, see the Amazon DynamoDB API Reference.

Description

The `DeleteTable` operation deletes a table and all of its items. After a `DeleteTable` request, the specified table is in the `DELETING` state until DynamoDB completes the deletion. If the table is in the `ACTIVE` state, you can delete it. If a table is in `CREATING` or `UPDATING` states, then DynamoDB returns a `ResourceInUseException` error. If the specified table does not exist, DynamoDB returns a `ResourceNotFoundException`. If table is already in the `DELETING` state, no error is returned.

Note

DynamoDB might continue to accept data plane operation requests, such as `GetItem` and `PutItem`, on a table in the `DELETING` state until the table deletion is complete.

Tables are unique among those associated with the AWS Account issuing the request, and the AWS region that receives the request (such as dynamodb.us-west-1.amazonaws.com). Each DynamoDB endpoint is entirely independent. For example, if you have two tables called "MyTable," one in dynamodb.us-west-2.amazonaws.com and one in dynamodb.us-west-1.amazonaws.com, they are completely independent and do not share any data; deleting one does not delete the other.

Use the DescribeTables operation to check the status of the table.

Requests

Syntax

```
// This header is abbreviated.
```

```
// For a sample of a complete header, see DynamoDB low-level API.
POST / HTTP/1.1
x-amz-target: DynamoDB_20111205.DeleteTable
content-type: application/x-amz-json-1.0

{"TableName":"Table1"}
```

Name	Description	Required
TableName	The name of the table to delete. Type: String	Yes

Responses

Syntax

```
HTTP/1.1 200 OK
x-amzn-RequestId: 4HONCKIVH1BFUDQ1U68CTG3N27VV4KQNSO5AEMVJF66Q9ASUAAJG
content-type: application/x-amz-json-1.0
content-length: 311
Date: Sun, 14 Aug 2011 22:56:22 GMT

{"TableDescription":
    {"CreationDateTime":1.313362508446E9,
    "KeySchema":
        {"HashKeyElement":{"AttributeName":"user","AttributeType":"S"},
        "RangeKeyElement":{"AttributeName":"time","AttributeType":"N"}},
    "ProvisionedThroughput":{"ReadCapacityUnits":10,"WriteCapacityUnits":10},
    "TableName":"Table1",
    "TableStatus":"DELETING"
    }
}
```

Name	Description
TableDescription	A container for the table properties.
CreationDateTime	Date when the table was created.

Name	Description
	Type: Number
KeySchema	The primary key (simple or composite) structure for the table. A name-value pair for the HashKeyElement is required, and a name-value pair for the RangeKeyElement is optional (only required for composite primary keys). For more information about primary keys, see Primary key. Type: Map of HashKeyElement , or HashKeyElement and RangeKeyElement for a composite primary key.
ProvisionedThroughput	Throughput for the specified table, consisting of values for ReadCapacityUnits and WriteCapacityUnits . See DynamoDB provisioned capacity mode.
ProvisionedThroughput : ReadCapacityUnits	The minimum number of ReadCapacityUnits consumed per second for the specified table before DynamoDB balances the load with other operations. Type: Number
ProvisionedThroughput :WriteCapacityUnits	The minimum number of WriteCapacityUnits consumed per second for the specified table before DynamoDB balances the load with other operations. Type: Number
TableName	The name of the deleted table. Type: String

Name	Description
TableStatus	The current state of the table (DELETING). Once the table is deleted, subsequent requests for the table return resource not found. Use the DescribeTables operation to check the status of the table. Type: String

Special errors

Error	Description
ResourceInUseException	Table is in state CREATING or UPDATING and can't be deleted.

Examples

Sample request

```
// This header is abbreviated. For a sample of a complete header, see DynamoDB low-level API.
POST / HTTP/1.1
x-amz-target: DynamoDB_20111205.DeleteTable
content-type: application/x-amz-json-1.0
content-length: 40

{"TableName":"favorite-movies-table"}
```

Sample response

```
HTTP/1.1 200 OK
x-amzn-RequestId: 4HONCKIVH1BFUDQ1U68CTG3N27VV4KQNSO5AEMVJF66Q9ASUAAJG
content-type: application/x-amz-json-1.0
content-length: 160
```

```
Date: Sun, 14 Aug 2011 17:20:03 GMT

{"TableDescription":
    {"CreationDateTime":1.313362508446E9,
    "KeySchema":
        {"HashKeyElement":{"AttributeName":"name","AttributeType":"S"}},
    "TableName":"favorite-movies-table",
    "TableStatus":"DELETING"
}
```

Related actions

- CreateTable
- DescribeTables

DescribeTables

> **Important**
>
> ***This section refers to API version 2011-12-05, which is deprecated and should not be used for new applications.***
>
> **For documentation on the current low-level API, see the Amazon DynamoDB API Reference.**

Description

Returns information about the table, including the current status of the table, the primary key schema and when the table was created. DescribeTable results are eventually consistent. If you use DescribeTable too early in the process of creating a table, DynamoDB returns a `ResourceNotFoundException`. If you use DescribeTable too early in the process of updating a table, the new values might not be immediately available.

Requests

Syntax

```
// This header is abbreviated.
// For a sample of a complete header, see DynamoDB low-level API.
```

```
POST / HTTP/1.1
x-amz-target: DynamoDB_20111205.DescribeTable
content-type: application/x-amz-json-1.0

{"TableName":"Table1"}
```

Name	Description	Required
TableName	The name of the table to describe. Type: String	Yes

Responses

Syntax

```
HTTP/1.1 200
x-amzn-RequestId: 8966d095-71e9-11e0-a498-71d736f27375
content-type: application/x-amz-json-1.0
Content-Length: 543


{"Table":
    {"CreationDateTime":1.309988345372E9,
    ItemCount:1,
    "KeySchema":
        {"HashKeyElement":{"AttributeName":"AttributeName1","AttributeType":"S"},
        "RangeKeyElement":{"AttributeName":"AttributeName2","AttributeType":"N"}},
    "ProvisionedThroughput":{"LastIncreaseDateTime": Date, "LastDecreaseDateTime":
 Date, "ReadCapacityUnits":10,"WriteCapacityUnits":10},
    "TableName":"Table1",
    "TableSizeBytes":1,
    "TableStatus":"ACTIVE"
    }
}
```

Name	Description
Table	Container for the table being described.

Name	Description
	Type: String
`CreationDateTime`	Date when the table was created in UNIX epoch time.
`ItemCount`	Number of items in the specified table. DynamoDB updates this value approximately every six hours. Recent changes might not be reflected in this value. Type: Number
`KeySchema`	The primary key (simple or composite) structure for the table. A name-value pair for the `HashKeyElement` is required, and a name-value pair for the `RangeKeyElement` is optional (only required for composite primary keys). The maximum hash key size is 2048 bytes. The maximum range key size is 1024 bytes. Both limits are enforced separatel y (i.e. you can have a combined hash + range 2048 + 1024 key). For more information about primary keys, see Primary key .
`ProvisionedThroughput`	Throughput for the specified table, consistin g of values for `LastIncreaseDateTime` (if applicable), `LastDecreaseDateTime` (if applicable), `ReadCapacityUnits` and `WriteCapacityUnits` . If the throughpu t for the table has never been increased or decreased, DynamoDB does not return values for those elements. See DynamoDB provision ed capacity mode. Type: Array

Name	Description
TableName	The name of the requested table. Type: String
TableSizeBytes	Total size of the specified table, in bytes. DynamoDB updates this value approximately every six hours. Recent changes might not be reflected in this value. Type: Number
TableStatus	The current state of the table (CREATING, ACTIVE, DELETING or UPDATING). Once the table is in the ACTIVE state, you can add data.

Special errors

No errors are specific to this operation.

Examples

The following examples show an HTTP POST request and response using the DescribeTable operation for a table named "comp-table". The table has a composite primary key.

Sample Request

```
// This header is abbreviated.
// For a sample of a complete header, see DynamoDB low-level API.
POST / HTTP/1.1
x-amz-target: DynamoDB_20111205.DescribeTable
content-type: application/x-amz-json-1.0

{"TableName":"users"}
```

Sample response

```
HTTP/1.1 200
x-amzn-RequestId: 8966d095-71e9-11e0-a498-71d736f27375
```

```
content-type: application/x-amz-json-1.0
content-length: 543

{"Table":
    {"CreationDateTime":1.309988345372E9,
    "ItemCount":23,
    "KeySchema":
        {"HashKeyElement":{"AttributeName":"user","AttributeType":"S"},
        "RangeKeyElement":{"AttributeName":"time","AttributeType":"N"}},
    "ProvisionedThroughput":{"LastIncreaseDateTime": 1.309988345384E9,
 "ReadCapacityUnits":10,"WriteCapacityUnits":10},
    "TableName":"users",
    "TableSizeBytes":949,
    "TableStatus":"ACTIVE"
    }
}
```

Related actions

- CreateTable
- DeleteTable
- ListTables

GetItem

> **Important**
>
> ***This section refers to API version 2011-12-05, which is deprecated and should not be used for new applications.***
>
> **For documentation on the current low-level API, see the Amazon DynamoDB API Reference.**

Description

The `GetItem` operation returns a set of `Attributes` for an item that matches the primary key. If there is no matching item, `GetItem` does not return any data.

The `GetItem` operation provides an eventually consistent read by default. If eventually consistent reads are not acceptable for your application, use `ConsistentRead`. Although this operation

might take longer than a standard read, it always returns the last updated value. For more information, see DynamoDB read consistency.

Requests

Syntax

```
// This header is abbreviated.
// For a sample of a complete header, see DynamoDB low-level API.
POST / HTTP/1.1
x-amz-target: DynamoDB_20111205.GetItem
content-type: application/x-amz-json-1.0

{"TableName":"Table1",
  "Key":
  {"HashKeyElement": {"S":"AttributeValue1"},
  "RangeKeyElement": {"N":"AttributeValue2"}
 },
 "AttributesToGet":["AttributeName3","AttributeName4"],
 "ConsistentRead":Boolean
}
```

Name	Description	Required
`TableName`	The name of the table containing the requested item. Type: String	Yes
`Key`	The primary key values that define the item. For more information about primary keys, see Primary key. Type: Map of `HashKeyElement` to its value and `RangeKeyElement` to its value.	Yes

Name	Description	Required
AttributesToGet	Array of Attribute names. If attribute names are not specified then all attributes will be returned. If some attributes are not found, they will not appear in the result. Type: Array	No
ConsistentRead	If set to true, then a consistent read is issued, otherwise eventually consistent is used. Type: Boolean	No

Responses

Syntax

```
HTTP/1.1 200
x-amzn-RequestId: 8966d095-71e9-11e0-a498-71d736f27375
content-type: application/x-amz-json-1.0
content-length: 144

{"Item":{
 "AttributeName3":{"S":"AttributeValue3"},
 "AttributeName4":{"N":"AttributeValue4"},
 "AttributeName5":{"B":"dmFsdWU="}
 },
"ConsumedCapacityUnits": 0.5
}
```

Name	Description
Item	Contains the requested attributes.

Name	Description
	Type: Map of attribute name-value pairs.
`ConsumedCapacityUnits`	The number of read capacity units consumed by the operation. This value shows the number applied toward your provisioned throughput. Requests for non-existent items consume the minimum read capacity units, depending on the type of read. For more information see DynamoDB provisioned capacity mode. Type: Number

Special errors

No errors specific to this operation.

Examples

For examples using the AWS SDK, see Working with items and attributes in DynamoDB.

Sample request

```
// This header is abbreviated.
// For a sample of a complete header, see DynamoDB low-level API.
POST / HTTP/1.1
x-amz-target: DynamoDB_20111205.GetItem
content-type: application/x-amz-json-1.0

{"TableName":"comptable",
 "Key":
  {"HashKeyElement":{"S":"Julie"},
  "RangeKeyElement":{"N":"1307654345"}},
 "AttributesToGet":["status","friends"],
 "ConsistentRead":true
}
```

Sample response

Notice the ConsumedCapacityUnits value is 1, because the optional parameter `ConsistentRead` is set to `true`. If `ConsistentRead` is set to `false` (or not specified) for the same request, the response is eventually consistent and the ConsumedCapacityUnits value would be 0.5.

```
HTTP/1.1 200
x-amzn-RequestId: 8966d095-71e9-11e0-a498-71d736f27375
content-type: application/x-amz-json-1.0
content-length: 72

{"Item":
 {"friends":{"SS":["Lynda, Aaron"]},
 "status":{"S":"online"}
 },
"ConsumedCapacityUnits": 1
}
```

ListTables

> **Important**
>
> ***This section refers to API version 2011-12-05, which is deprecated and should not be used for new applications.***
>
> **For documentation on the current low-level API, see the Amazon DynamoDB API Reference.**

Description

Returns an array of all the tables associated with the current account and endpoint.

Each DynamoDB endpoint is entirely independent. For example, if you have two tables called "MyTable," one in dynamodb.us-west-2.amazonaws.com and one in dynamodb.us-east-1.amazonaws.com, they are completely independent and do not share any data. The ListTables operation returns all of the table names associated with the account making the request, for the endpoint that receives the request.

Requests

Syntax

```
// This header is abbreviated.
// For a sample of a complete header, see DynamoDB low-level API.
POST / HTTP/1.1
x-amz-target: DynamoDB_20111205.ListTables
content-type: application/x-amz-json-1.0

{"ExclusiveStartTableName":"Table1","Limit":3}
```

The ListTables operation, by default, requests all of the table names associated with the account making the request, for the endpoint that receives the request.

Name	Description	Required
`Limit`	A number of maximum table names to return. Type: Integer	No
`ExclusiveStartTableName`	The name of the table that starts the list. If you already ran a ListTables operation and received an `LastEvaluatedTableName` value in the response, use that value here to continue the list. Type: String	No

Responses

Syntax

```
HTTP/1.1 200 OK
x-amzn-RequestId: S1LEK2DPQP8OJNHVHL8OU2M7KRVV4KQNSO5AEMVJF66Q9ASUAAJG
content-type: application/x-amz-json-1.0
```

```
content-length: 81
Date: Fri, 21 Oct 2011 20:35:38 GMT

{"TableNames":["Table1","Table2","Table3"], "LastEvaluatedTableName":"Table3"}
```

Name	Description
`TableNames`	The names of the tables associated with the current account at the current endpoint. Type: Array
`LastEvaluatedTableName`	The name of the last table in the current list, only if some tables for the account and endpoint have not been returned. This value does not exist in a response if all table names are already returned. Use this value as the `ExclusiveStartTableName` in a new request to continue the list until all the table names are returned. Type: String

Special errors

No errors are specific to this operation.

Examples

The following examples show an HTTP POST request and response using the ListTables operation.

Sample request

```
// This header is abbreviated.
// For a sample of a complete header, see DynamoDB low-level API.
POST / HTTP/1.1
x-amz-target: DynamoDB_20111205.ListTables
content-type: application/x-amz-json-1.0
```

```
{"ExclusiveStartTableName":"comp2","Limit":3}
```

Sample response

```
HTTP/1.1 200 OK
x-amzn-RequestId: S1LEK2DPQP8OJNHVHL8OU2M7KRVV4KQNSO5AEMVJF66Q9ASUAAJG
content-type: application/x-amz-json-1.0
content-length: 81
Date: Fri, 21 Oct 2011 20:35:38 GMT

{"LastEvaluatedTableName":"comp5","TableNames":["comp3","comp4","comp5"]}
```

Related actions

- DescribeTables
- CreateTable
- DeleteTable

PutItem

> **Important**
>
> ***This section refers to API version 2011-12-05, which is deprecated and should not be used for new applications.***
> **For documentation on the current low-level API, see the Amazon DynamoDB API Reference.**

Description

Creates a new item, or replaces an old item with a new item (including all the attributes). If an item already exists in the specified table with the same primary key, the new item completely replaces the existing item. You can perform a conditional put (insert a new item if one with the specified primary key doesn't exist), or replace an existing item if it has certain attribute values.

Attribute values may not be null; string and binary type attributes must have lengths greater than zero; and set type attributes must not be empty. Requests with empty values will be rejected with a `ValidationException`.

> **Note**
>
> To ensure that a new item does not replace an existing item, use a conditional put operation with `Exists` set to `false` for the primary key attribute, or attributes.

For more information about using `PutItem`, see Working with items and attributes in DynamoDB.

Requests

Syntax

```
// This header is abbreviated.
// For a sample of a complete header, see DynamoDB low-level API.
POST / HTTP/1.1
x-amz-target: DynamoDB_20111205.PutItem
content-type: application/x-amz-json-1.0

{"TableName":"Table1",
    "Item":{
        "AttributeName1":{"S":"AttributeValue1"},
        "AttributeName2":{"N":"AttributeValue2"},
        "AttributeName5":{"B":"dmFsdWU="}
    },
    "Expected":{"AttributeName3":{"Value": {"S":"AttributeValue"}, "Exists":Boolean}},
    "ReturnValues":"ReturnValuesConstant"}
```

Name	Description	Required
`TableName`	The name of the table to contain the item. Type: String	Yes
`Item`	A map of the attributes for the item, and must include the primary key values that define the item. Other attribute name-value pairs can be provided for the item.	Yes

Name	Description	Required
	For more information about primary keys, see Primary key. Type: Map of attribute names to attribute values.	
`Expected`	Designates an attribute for a conditional put. The `Expected` parameter allows you to provide an attribute name, and whether or not DynamoDB should check to see if the attribute value already exists; or if the attribute value exists and has a particular value before changing it. Type: Map of an attribute names to an attribute value, and whether it exists.	No
`Expected:AttributeName`	The name of the attribute for the conditional put. Type: String	No

Name	Description	Required

`Expected:AttributeName: ExpectedAttributeValue`	Use this parameter to specify whether or not a value already exists for the attribute name-value pair.
	
	The following JSON notation replaces the item if the "Color" attribute doesn't already exist for that item:
	
	`"Expected" : {"Color":{"Exists":false}}`
	
	The following JSON notation checks to see if the attribute with name "Color" has an existing value of "Yellow" before replacing the item:
	
	`"Expected" : {"Color":{"Exists":true,{"Value":{"S":"Yellow"}}}}`
	
	By default, if you use the `Expected` parameter and provide a `Value`, DynamoDB assumes the attribute exists and has a current value to be replaced. So you don't have to specify `{"Exists":true}`, because it is implied. You can shorten the request to:
	
	`"Expected" :`
		No

Name	Description	Required
	`{"Color":{"Value":{"S":"Yellow"}}}` **Note** If you specify `{"Exists":true}` without an attribute value to check, DynamoDB returns an error.	
`ReturnValues`	Use this parameter if you want to get the attribute name-value pairs before they were updated with the `PutItem` request. Possible parameter values are `NONE` (default) or `ALL_OLD`. If `ALL_OLD` is specified, and `PutItem` overwrote an attribute name-value pair, the content of the old item is returned. If this parameter is not provided or is `NONE`, nothing is returned. Type: String	No

Responses

Syntax

The following syntax example assumes the request specified a `ReturnValues` parameter of `ALL_OLD`; otherwise, the response has only the `ConsumedCapacityUnits` element.

```
HTTP/1.1 200
x-amzn-RequestId: 8966d095-71e9-11e0-a498-71d736f27375
content-type: application/x-amz-json-1.0
content-length: 85

{"Attributes":
 {"AttributeName3":{"S":"AttributeValue3"},
 "AttributeName2":{"SS":"AttributeValue2"},
 "AttributeName1":{"SS":"AttributeValue1"},
 },
"ConsumedCapacityUnits":1
}
```

Name	Description
`Attributes`	Attribute values before the put operation, but only if the `ReturnValues` parameter is specified as `ALL_OLD` in the request. Type: Map of attribute name-value pairs.
`ConsumedCapacityUnits`	The number of write capacity units consumed by the operation. This value shows the number applied toward your provision ed throughput. For more information see DynamoDB provisioned capacity mode. Type: Number

Special errors

Error	Description
`ConditionalCheckFailedException`	Conditional check failed. An expected attribute value was not found.
`ResourceNotFoundException`	The specified item or attribute was not found.

Examples

For examples using the AWS SDK, see Working with items and attributes in DynamoDB.

Sample request

```
// This header is abbreviated. For a sample of a complete header, see DynamoDB low-level API.
POST / HTTP/1.1
x-amz-target: DynamoDB_20111205.PutItem
content-type: application/x-amz-json-1.0

{"TableName":"comp5",
 "Item":
  {"time":{"N":"300"},
  "feeling":{"S":"not surprised"},
  "user":{"S":"Riley"}
  },
 "Expected":
  {"feeling":{"Value":{"S":"surprised"},"Exists":true}}
 "ReturnValues":"ALL_OLD"
}
```

Sample response

```
HTTP/1.1 200
x-amzn-RequestId: 8952fa74-71e9-11e0-a498-71d736f27375
content-type: application/x-amz-json-1.0
content-length: 84

{"Attributes":
 {"feeling":{"S":"surprised"},
 "time":{"N":"300"},
 "user":{"S":"Riley"}},
"ConsumedCapacityUnits":1
}
```

Related actions

- UpdateItem
- DeleteItem
- GetItem

- BatchGetItem

Query

> **Important**
>
> ***This section refers to API version 2011-12-05, which is deprecated and should not be used for new applications.***
>
> **For documentation on the current low-level API, see the Amazon DynamoDB API Reference.**

Description

A `Query` operation gets the values of one or more items and their attributes by primary key (`Query` is only available for hash-and-range primary key tables). You must provide a specific `HashKeyValue`, and can narrow the scope of the query using comparison operators on the `RangeKeyValue` of the primary key. Use the `ScanIndexForward` parameter to get results in forward or reverse order by range key.

Queries that do not return results consume the minimum read capacity units according to the type of read.

> **Note**
>
> If the total number of items meeting the query parameters exceeds the 1MB limit, the query stops and results are returned to the user with a `LastEvaluatedKey` to continue the query in a subsequent operation. Unlike a Scan operation, a Query operation never returns an empty result set *and* a `LastEvaluatedKey`. The `LastEvaluatedKey` is only provided if the results exceed 1MB, or if you have used the `Limit` parameter.
> The result can be set for a consistent read using the `ConsistentRead` parameter.

Requests

Syntax

```
// This header is abbreviated.
```

```
// For a sample of a complete header, see DynamoDB low-level API.
POST / HTTP/1.1
x-amz-target: DynamoDB_20111205.Query
content-type: application/x-amz-json-1.0

{"TableName":"Table1",
 "Limit":2,
 "ConsistentRead":true,
 "HashKeyValue":{"S":"AttributeValue1":},
 "RangeKeyCondition": {"AttributeValueList":
[{"N":"AttributeValue2"}],"ComparisonOperator":"GT"}
 "ScanIndexForward":true,
 "ExclusiveStartKey":{
  "HashKeyElement":{"S":"AttributeName1"},
  "RangeKeyElement":{"N":"AttributeName2"}
 },
    "AttributesToGet":["AttributeName1", "AttributeName2", "AttributeName3"]},
}
```

Name	Description	Required
`TableName`	The name of the table containing the requested items. Type: String	Yes
`AttributesToGet`	Array of Attribute names. If attribute names are not specified then all attributes will be returned. If some attributes are not found, they will not appear in the result. Type: Array	No
`Limit`	The maximum number of items to return (not necessarily the number of matching items). If Dynamo	No

Name	Description	Required
	DB processes the number of items up to the limit while querying the table, it stops the query and returns the matching values up to that point, and a `LastEvaluatedKey` to apply in a subsequent operation to continue the query. Also, if the result set size exceeds 1MB before DynamoDB hits this limit, it stops the query and returns the matching values, and a `LastEvaluatedKey` to apply in a subsequent operation to continue the query. Type: Number	
`ConsistentRead`	If set to `true`, then a consistent read is issued, otherwise eventually consistent is used. Type: Boolean	No

Name	Description	Required
`Count`	If set to `true`, DynamoDB returns a total number of items that match the query parameters, instead of a list of the matching items and their attributes. You can apply the `Limit` parameter to count-only queries. Do not set `Count` to `true` while providing a list of `AttributesToGet` ; otherwise, DynamoDB returns a validation error. For more information, see Counting the items in the results. Type: Boolean	No
`HashKeyValue`	Attribute value of the hash component of the composite primary key. Type: String, Number, or Binary	Yes

Name	Description	Required
RangeKeyCondition	A container for the attribute values and comparison operators to use for the query. A query request does not require a RangeKeyCondition. If you provide only the HashKeyValue, DynamoDB returns all items with the specified hash key element value. Type: Map	No
RangeKeyCondition : AttributeValueList	The attribute values to evaluate for the query parameters. The AttributeValueList contains one attribute value, unless a BETWEEN comparison is specified. For the BETWEEN comparison, the AttributeValueList contains two attribute values. Type: A map of AttributeValue to a ComparisonOperator.	No

Name	Description	Required
`RangeKeyCondition : ComparisonOperator`	The criteria for evaluating the provided attributes, such as equals, greater-than, etc. The following are valid comparison operators for a Query operation. **Note** String value comparisons for greater than, equals, or less than are based on ASCII character code values. For example, a is greater than A, and aa is greater than B. For a list of code values, see http://en.wikipedia.org/wiki/ASCII#ASCII_printable_characters. For Binary, DynamoDB treats each byte of the binary data as unsigned when it compares binary values, for example when evaluating query expressions. Type: String or Binary	No

Name	Description	Required
	EQ : Equal. For EQ, `AttributeValueList` can contain only one `AttributeValue` of type String, Number, or Binary (not a set). If an item contains an `AttributeValue` of a different type than the one specified in the request, the value does not match. For example, `{"S":"6"}` does not equal `{"N":"6"}` . Also, `{"N":"6"}` does not equal `{"NS":["6", "2", "1"]}`.	
	LE : Less than or equal. For LE, `AttributeValueList` can contain only one `AttributeValue` of type String, Number, or Binary (not a set). If an item contains an `AttributeValue` of a different type than the one specified in the request, the value does not match. For example, `{"S":"6"}` does not equal `{"N":"6"}` . Also, `{"N":"6"}` does not compare to `{"NS":["6", "2", "1"]}`.	

Name	Description	Required
	LT : Less than. For LT, `AttributeValueList` can contain only one `AttributeValue` of type String, Number, or Binary (not a set). If an item contains an `AttributeValue` of a different type than the one specified in the request, the value does not match. For example, `{"S":"6"}` does not equal `{"N":"6"}`. Also, `{"N":"6"}` does not compare to `{"NS":["6", "2", "1"]}`.	
	GE : Greater than or equal. For GE, `AttributeValueList` can contain only one `AttributeValue` of type String, Number, or Binary (not a set). If an item contains an `AttributeValue` of a different type than the one specified in the request, the value does not match. For example, `{"S":"6"}` does not equal `{"N":"6"}`. Also, `{"N":"6"}` does not compare to `{"NS":["6", "2", "1"]}`.	

Name	Description	Required
	GT : Greater than. For GT, `AttributeValueList` can contain only one `AttributeValue` of type String, Number, or Binary (not a set). If an item contains an `AttributeValue` of a different type than the one specified in the request, the value does not match. For example, `{"S":"6"}` does not equal `{"N":"6"}` . Also, `{"N":"6"}` does not compare to `{"NS":["6", "2", "1"]}`.	
	`BEGINS_WITH` : checks for a prefix. For `BEGINS_WITH` , `AttributeValueList` can contain only one `AttributeValue` of type String or Binary (not a Number or a set). The target attribute of the comparison must be a String or Binary (not a Number or a set).	

Name	Description	Required
	BETWEEN : Greater than, or equal to, the first value and less than, or equal to, the second value. For BETWEEN, AttributeValueList must contain two AttributeValue elements of the same type, either String, Number, or Binary (not a set). A target attribute matches if the target value is greater than, or equal to, the first element and less than, or equal to, the second element. If an item contains an AttributeValue of a different type than the one specified in the request, the value does not match. For example, {"S":"6"} does not compare to {"N":"6"} . Also, {"N":"6"} does not compare to {"NS":["6", "2", "1"]}.	

Name	Description	Required
ScanIndexForward	Specifies ascending or descending traversal of the index. DynamoDB returns results reflecting the requested order determined by the range key: If the data type is Number, the results are returned in numeric order; otherwise, the traversal is based on ASCII character code values. Type: Boolean Default is true (ascending).	No
ExclusiveStartKey	Primary key of the item from which to continue an earlier query. An earlier query might provide this value as the LastEvaluatedKey if that query operation was interrupted before completing the query; either because of the result set size or the Limit parameter. The LastEvaluatedKey can be passed back in a new query request to continue the operation from that point. Type: HashKeyElement, or HashKeyElement and RangeKeyElement for a composite primary key.	No

Responses

Syntax

```
HTTP/1.1 200
x-amzn-RequestId: 8966d095-71e9-11e0-a498-71d736f27375
content-type: application/x-amz-json-1.0
content-length: 308

{"Count":2,"Items":[{
    "AttributeName1":{"S":"AttributeValue1"},
    "AttributeName2":{"N":"AttributeValue2"},
    "AttributeName3":{"S":"AttributeValue3"}
    },{
    "AttributeName1":{"S":"AttributeValue3"},
    "AttributeName2":{"N":"AttributeValue4"},
    "AttributeName3":{"S":"AttributeValue3"},
    "AttributeName5":{"B":"dmFsdWU="}
}],
    "LastEvaluatedKey":{"HashKeyElement":{"AttributeValue3":"S"},
                        "RangeKeyElement":{"AttributeValue4":"N"}
     },
     "ConsumedCapacityUnits":1
}
```

Name	Description
`Items`	Item attributes meeting the query parameters. Type: Map of attribute names to and their data types and values.
`Count`	Number of items in the response. For more information, see Counting the items in the results. Type: Number
`LastEvaluatedKey`	Primary key of the item where the query operation stopped, inclusive of the previous result set. Use this value to start a new

Name	Description
	operation excluding this value in the new request. The `LastEvaluatedKey` is `null` when the entire query result set is complete (i.e. the operation processed the "last page"). Type: `HashKeyElement`, or `HashKeyElement` and `RangeKeyElement` for a composite primary key.
`ConsumedCapacityUnits`	The number of read capacity units consumed by the operation. This value shows the number applied toward your provisioned throughput. For more information see DynamoDB provisioned capacity mode. Type: Number

Special errors

Error	Description
`ResourceNotFoundException`	The specified table was not found.

Examples

For examples using the AWS SDK, see Querying tables in DynamoDB.

Sample request

```
// This header is abbreviated. For a sample of a complete header, see DynamoDB low-level API.
POST / HTTP/1.1
x-amz-target: DynamoDB_20111205.Query
content-type: application/x-amz-json-1.0
```

```
{"TableName":"1-hash-rangetable",
 "Limit":2,
 "HashKeyValue":{"S":"John"},
 "ScanIndexForward":false,
 "ExclusiveStartKey":{
  "HashKeyElement":{"S":"John"},
  "RangeKeyElement":{"S":"The Matrix"}
 }
}
```

Sample response

```
HTTP/1.1 200
x-amzn-RequestId: 3647e778-71eb-11e0-a498-71d736f27375
content-type: application/x-amz-json-1.0
content-length: 308

{"Count":2,"Items":[{
 "fans":{"SS":["Jody","Jake"]},
 "name":{"S":"John"},
 "rating":{"S":"***"},
 "title":{"S":"The End"}
 },{
 "fans":{"SS":["Jody","Jake"]},
 "name":{"S":"John"},
 "rating":{"S":"***"},
 "title":{"S":"The Beatles"}
 }],
 "LastEvaluatedKey":{"HashKeyElement":{"S":"John"},"RangeKeyElement":{"S":"The
 Beatles"}},
"ConsumedCapacityUnits":1
}
```

Sample request

```
// This header is abbreviated. For a sample of a complete header, see DynamoDB low-
level API.
POST / HTTP/1.1
x-amz-target: DynamoDB_20111205.Query
content-type: application/x-amz-json-1.0
```

```
{"TableName":"1-hash-rangetable",
 "Limit":2,
 "HashKeyValue":{"S":"Airplane"},
 "RangeKeyCondition":{"AttributeValueList":[{"N":"1980"}],"ComparisonOperator":"EQ"},
 "ScanIndexForward":false}
```

Sample response

```
HTTP/1.1 200
x-amzn-RequestId: 8b9ee1ad-774c-11e0-9172-d954e38f553a
content-type: application/x-amz-json-1.0
content-length: 119

{"Count":1,"Items":[{
 "fans":{"SS":["Dave","Aaron"]},
 "name":{"S":"Airplane"},
 "rating":{"S":"***"},
 "year":{"N":"1980"}
 }],
"ConsumedCapacityUnits":1
}
```

Related actions

- Scan

Scan

> **Important**
>
> ***This section refers to API version 2011-12-05, which is deprecated and should not be used for new applications.***
>
> **For documentation on the current low-level API, see the Amazon DynamoDB API Reference.**

Description

The `Scan` operation returns one or more items and its attributes by performing a full scan of a table. Provide a `ScanFilter` to get more specific results.

Note

If the total number of scanned items exceeds the 1MB limit, the scan stops and results are returned to the user with a `LastEvaluatedKey` to continue the scan in a subsequent operation. The results also include the number of items exceeding the limit. A scan can result in no table data meeting the filter criteria.
The result set is eventually consistent.

Requests

Syntax

```
// This header is abbreviated.
// For a sample of a complete header, see DynamoDB low-level API.
POST / HTTP/1.1
x-amz-target: DynamoDB_20111205.Scan
content-type: application/x-amz-json-1.0

{"TableName":"Table1",
    "Limit": 2,
    "ScanFilter":{
        "AttributeName1":{"AttributeValueList":
[{"S":"AttributeValue"}],"ComparisonOperator":"EQ"}
    },
    "ExclusiveStartKey":{
        "HashKeyElement":{"S":"AttributeName1"},
        "RangeKeyElement":{"N":"AttributeName2"}
    },
    "AttributesToGet":["AttributeName1", "AttributeName2", "AttributeName3"]},
}
```

Name	Description	Required
`TableName`	The name of the table containing the requested items. Type: String	Yes

Name	Description	Required
`AttributesToGet`	Array of Attribute names. If attribute names are not specified then all attributes will be returned. If some attributes are not found, they will not appear in the result. Type: Array	No
`Limit`	The maximum number of items to evaluate (not necessarily the number of matching items). If DynamoDB processes the number of items up to the limit while processing the results, it stops and returns the matching values up to that point, and a `LastEvaluatedKey` to apply in a subsequent operation to continue retrieving items. Also, if the scanned data set size exceeds 1MB before DynamoDB reaches this limit, it stops the scan and returns the matching values up to the limit, and a `LastEvaluatedKey` to apply in a subsequent operation to continue the scan. Type: Number	No

Name	Description	Required
`Count`	If set to `true`, DynamoDB returns a total number of items for the Scan operation, even if the operation has no matching items for the assig ned filter. You can apply the Limit parameter to count-onl y scans. Do not set `Count` to `true` while providing a list of `AttributesToGet` , otherwise DynamoDB returns a validation error. For more information, see Counting the items in the results. Type: Boolean	No
`ScanFilter`	Evaluates the scan results and returns only the desired values. Multiple condition s are treated as "AND" operations: all conditions must be met to be included in the results. Type: A map of attribute names to values with comparison operators.	No

Name	Description	Required
`ScanFilter :AttributeValueList`	The values and conditions to evaluate the scan results for the filter. Type: A map of `AttributeValue` to a `Condition` .	No

Name	Description	Required
`ScanFilter : ComparisonOperator`	The criteria for evaluating the provided attributes, such as equals, greater-than, etc. The following are valid comparison operators for a scan operation. **Note** String value comparisons for greater than, equals, or less than are based on ASCII character code values. For example, a is greater than A, and aa is greater than B. For a list of code values, see http://en.wikipedia.org/wiki/ASCII#ASCII_printable_characters. For Binary, DynamoDB treats each byte of the binary data as unsigned when it compares binary values, for example when evaluating query expressions. Type: String or Binary	No

Name	Description	Required
	EQ : Equal. For EQ, `AttributeValueList` can contain only one `AttributeValue` of type String, Number, or Binary (not a set). If an item contains an `AttributeValue` of a different type than the one specified in the request, the value does not match. For example, `{"S":"6"}` does not equal `{"N":"6"}`. Also, `{"N":"6"}` does not equal `{"NS":["6", "2", "1"]}`.	
	NE : Not Equal. For NE, `AttributeValueList` can contain only one `AttributeValue` of type String, Number, or Binary (not a set). If an item contains an `AttributeValue` of a different type than the one specified in the request, the value does not match. For example, `{"S":"6"}` does not equal `{"N":"6"}`. Also, `{"N":"6"}` does not equal `{"NS":["6", "2", "1"]}`.	

Name	Description	Required
	LE : Less than or equal. For LE, Attribute ValueList can contain only one AttributeValue of type String, Number, or Binary (not a set). If an item contains an Attribute Value of a different type than the one specified in the request, the value does not match. For example, {"S":"6"} does not equal {"N":"6"} . Also, {"N":"6"} does not compare to {"NS":["6", "2", "1"]}.	
	LT : Less than. For LT, Attribute ValueList can contain only one AttributeValue of type String, Number, or Binary (not a set). If an item contains an Attribute Value of a different type than the one specified in the request, the value does not match. For example, {"S":"6"} does not equal {"N":"6"} . Also, {"N":"6"} does not compare to {"NS":["6", "2", "1"]}.	

Name	Description	Required
	GE : Greater than or equal. For GE, `AttributeValueList` can contain only one `AttributeValue` of type String, Number, or Binary (not a set). If an item contains an `AttributeValue` of a different type than the one specified in the request, the value does not match. For example, `{"S":"6"}` does not equal `{"N":"6"}`. Also, `{"N":"6"}` does not compare to `{"NS":["6", "2", "1"]}`.	
	GT : Greater than. For GT, `AttributeValueList` can contain only one `AttributeValue` of type String, Number, or Binary (not a set). If an item contains an `AttributeValue` of a different type than the one specified in the request, the value does not match. For example, `{"S":"6"}` does not equal `{"N":"6"}`. Also, `{"N":"6"}` does not compare to `{"NS":["6", "2", "1"]}`.	

Name	Description	Required
	NOT_NULL : Attribute exists.	
	NULL : Attribute does not exist.	
	CONTAINS : checks for a subsequence, or value in a set. For CONTAINS, AttributeValueList can contain only one AttributeValue of type String, Number, or Binary (not a set). If the target attribute of the comparison is a String, then the operation checks for a substring match. If the target attribute of the comparison is Binary, then the operation looks for a subsequence of the target th at matches the input. If the target attribute of the compa rison is a set ("SS", "NS", or "BS"), then the operation checks for a member of the set (not as a substring).	

Name	Description	Required
	`NOT_CONTAINS` : checks for absence of a subsequence, or absence of a value in a set. For `NOT_CONTAINS` , `AttributeValueList` can contain only one `AttributeValue` of type String, Number, or Binary (not a set). If the target attribute of the comparison is a String, then the operation checks for the absence of a substring match. If the target attribute of the comparison is Binary, then the operation checks for the absence of a subsequence of the target that matches the input. If the target attribute of the comparison is a set ("SS", "NS", or "BS"), then the operation checks for the absence of a member of the set (not as a substring).	

Name	Description	Required
	BEGINS_WITH : checks for a prefix. For BEGINS_WITH , AttributeValueList can contain only one AttributeValue of type String or Binary (not a Number or a set). The target attribute of the comparison must be a String or Binary (not a Number or a set).	
	IN : checks for exact matches. For IN, AttributeValueList can contain more than one AttributeValue of type String, Number, or Binary (not a set). The target attribute of the comparison must be of the same type and exact value to match. A String never matches a String set.	

Name	Description	Required
	BETWEEN : Greater than, or equal to, the first value and less than, or equal to, the second value. For BETWEEN, AttributeValueList must contain two AttributeValue elements of the same type, either String, Number, or Binary (not a set). A target attribute matches if the target value is greater than, or equal to, the first element and less than, or equal to, the second element. If an item contains an AttributeValue of a different type than the one specified in the request, the value does not match. For example, {"S":"6"} does not compare to {"N":"6"} . Also, {"N":"6"} does not compare to {"NS":["6", "2", "1"]}.	

Name	Description	Required
ExclusiveStartKey	Primary key of the item from which to continue an earlier scan. An earlier scan might provide this value if that scan operation was interrupted before scanning the entire table; either because of the result set size or the Limit parameter. The LastEvaluatedKey can be passed back in a new scan request to continue the operation from that point. Type: HashKeyElement, or HashKeyElement and RangeKeyElement for a composite primary key.	No

Responses

Syntax

```
HTTP/1.1 200
x-amzn-RequestId: 8966d095-71e9-11e0-a498-71d736f27375
content-type: application/x-amz-json-1.0
content-length: 229

{"Count":2,"Items":[{
    "AttributeName1":{"S":"AttributeValue1"},
    "AttributeName2":{"S":"AttributeValue2"},
    "AttributeName3":{"S":"AttributeValue3"}
    },{
    "AttributeName1":{"S":"AttributeValue4"},
    "AttributeName2":{"S":"AttributeValue5"},
    "AttributeName3":{"S":"AttributeValue6"},
    "AttributeName5":{"B":"dmFsdWU="}
```

```
    }],
    "LastEvaluatedKey":
        {"HashKeyElement":{"S":"AttributeName1"},
        "RangeKeyElement":{"N":"AttributeName2"},
    "ConsumedCapacityUnits":1,
    "ScannedCount":2}
}
```

Name	Description
`Items`	Container for the attributes meeting the operation parameters. Type: Map of attribute names to and their data types and values.
`Count`	Number of items in the response. For more information, see Counting the items in the results. Type: Number
`ScannedCount`	Number of items in the complete scan before any filters are applied. A high `ScannedCount` value with few, or no, `Count` results indicates an inefficient `Scan` operation. For more information, see Counting the items in the results. Type: Number
`LastEvaluatedKey`	Primary key of the item where the scan operation stopped. Provide this value in a subsequent scan operation to continue the operation from that point. The `LastEvaluatedKey` is `null` when the entire scan result set is complete (i.e. the operation processed the “last page”).

Name	Description
ConsumedCapacityUnits	The number of read capacity units consumed by the operation. This value shows the number applied toward your provision ed throughput. For more information see DynamoDB provisioned capacity mode. Type: Number

Special errors

Error	Description
ResourceNotFoundException	The specified table was not found.

Examples

For examples using the AWS SDK, see Scanning tables in DynamoDB.

Sample request

```
// This header is abbreviated. For a sample of a complete header, see DynamoDB low-level API.
POST / HTTP/1.1
x-amz-target: DynamoDB_20111205.Scan
content-type: application/x-amz-json-1.0

{"TableName":"1-hash-rangetable","ScanFilter":{}}
```

Sample response

```
HTTP/1.1 200
x-amzn-RequestId: 4e8a5fa9-71e7-11e0-a498-71d736f27375
content-type: application/x-amz-json-1.0
content-length: 465

{"Count":4,"Items":[{
```

```
 "date":{"S":"1980"},
 "fans":{"SS":["Dave","Aaron"]},
 "name":{"S":"Airplane"},
 "rating":{"S":"***"}
 },{
 "date":{"S":"1999"},
 "fans":{"SS":["Ziggy","Laura","Dean"]},
 "name":{"S":"Matrix"},
 "rating":{"S":"*****"}
 },{
 "date":{"S":"1976"},
 "fans":{"SS":["Riley"]},"
 name":{"S":"The Shaggy D.A."},
 "rating":{"S":"**"}
 },{
 "date":{"S":"1985"},
 "fans":{"SS":["Fox","Lloyd"]},
 "name":{"S":"Back To The Future"},
 "rating":{"S":"****"}
 }],
    "ConsumedCapacityUnits":0.5
 "ScannedCount":4}
```

Sample request

```
// This header is abbreviated. For a sample of a complete header, see DynamoDB low-
level API.
POST / HTTP/1.1
x-amz-target: DynamoDB_20111205.Scan
content-type: application/x-amz-json-1.0
content-length: 125

{"TableName":"comp5",
 "ScanFilter":
  {"time":
   {"AttributeValueList":[{"N":"400"}],
   "ComparisonOperator":"GT"}
 }
}
```

Sample response

```
HTTP/1.1 200 OK
```

```
x-amzn-RequestId: PD1CQK9QCTERLTJP20VALJ60TRVV4KQNSO5AEMVJF66Q9ASUAAJG
content-type: application/x-amz-json-1.0
content-length: 262
Date: Mon, 15 Aug 2011 16:52:02 GMT

{"Count":2,
 "Items":[
  {"friends":{"SS":["Dave","Ziggy","Barrie"]},
  "status":{"S":"chatting"},
  "time":{"N":"2000"},
  "user":{"S":"Casey"}},
  {"friends":{"SS":["Dave","Ziggy","Barrie"]},
  "status":{"S":"chatting"},
  "time":{"N":"2000"},
  "user":{"S":"Fredy"}
  }],
"ConsumedCapacityUnits":0.5
"ScannedCount":4
}
```

Sample request

```
// This header is abbreviated. For a sample of a complete header, see DynamoDB low-
level API.
POST / HTTP/1.1
x-amz-target: DynamoDB_20111205.Scan
content-type: application/x-amz-json-1.0

{"TableName":"comp5",
 "Limit":2,
 "ScanFilter":
  {"time":
   {"AttributeValueList":[{"N":"400"}],
   "ComparisonOperator":"GT"}
 },
 "ExclusiveStartKey":
  {"HashKeyElement":{"S":"Fredy"},"RangeKeyElement":{"N":"2000"}}
}
```

Sample response

```
HTTP/1.1 200 OK
x-amzn-RequestId: PD1CQK9QCTERLTJP20VALJ60TRVV4KQNSO5AEMVJF66Q9ASUAAJG
```

```
content-type: application/x-amz-json-1.0
content-length: 232
Date: Mon, 15 Aug 2011 16:52:02 GMT

{"Count":1,
 "Items":[
  {"friends":{"SS":["Jane","James","John"]},
  "status":{"S":"exercising"},
  "time":{"N":"2200"},
  "user":{"S":"Roger"}}
 ],
 "LastEvaluatedKey":{"HashKeyElement":{"S":"Riley"},"RangeKeyElement":{"N":"250"}},
"ConsumedCapacityUnits":0.5
"ScannedCount":2
}
```

Related actions

- Query
- BatchGetItem

UpdateItem

> **Important**
>
> ***This section refers to API version 2011-12-05, which is deprecated and should not be used for new applications.***
>
> **For documentation on the current low-level API, see the Amazon DynamoDB API Reference.**

Description

Edits an existing item's attributes. You can perform a conditional update (insert a new attribute name-value pair if it doesn't exist, or replace an existing name-value pair if it has certain expected attribute values).

> **Note**
>
> You cannot update the primary key attributes using UpdateItem. Instead, delete the item and use PutItem to create a new item with new attributes.

The UpdateItem operation includes an `Action` parameter, which defines how to perform the update. You can put, delete, or add attribute values.

Attribute values may not be null; string and binary type attributes must have lengths greater than zero; and set type attributes must not be empty. Requests with empty values will be rejected with a `ValidationException`.

If an existing item has the specified primary key:

- **PUT—** Adds the specified attribute. If the attribute exists, it is replaced by the new value.
- **DELETE—** If no value is specified, this removes the attribute and its value. If a set of values is specified, then the values in the specified set are removed from the old set. So if the attribute value contains [a,b,c] and the delete action contains [a,c], then the final attribute value is [b]. The type of the specified value must match the existing value type. Specifying an empty set is not valid.
- **ADD—** Only use the add action for numbers or if the target attribute is a set (including string sets). ADD does not work if the target attribute is a single string value or a scalar binary value. The specified value is added to a numeric value (incrementing or decrementing the existing numeric value) or added as an additional value in a string set. If a set of values is specified, the values are added to the existing set. For example if the original set is [1,2] and supplied value is [3], then after the add operation the set is [1,2,3], not [4,5]. An error occurs if an Add action is specified for a set attribute and the attribute type specified does not match the existing set type.

  If you use ADD for an attribute that does not exist, the attribute and its values are added to the item.

If no item matches the specified primary key:

- **PUT—** Creates a new item with specified primary key. Then adds the specified attribute.
- **DELETE—** Nothing happens.

- **ADD—** Creates an item with supplied primary key and number (or set of numbers) for the attribute value. Not valid for a string or a binary type.

> **Note**
>
> If you use ADD to increment or decrement a number value for an item that doesn't exist before the update, DynamoDB uses `0` as the initial value. Also, if you update an item using ADD to increment or decrement a number value for an attribute that doesn't exist before the update (but the item does) DynamoDB uses `0` as the initial value. For example, you use ADD to add `+3` to an attribute that did not exist before the update. DynamoDB uses `0` for the initial value, and the value after the update is `3`.

For more information about using this operation, see Working with items and attributes in DynamoDB.

Requests

Syntax

```
// This header is abbreviated.
// For a sample of a complete header, see DynamoDB low-level API.
POST / HTTP/1.1
x-amz-target: DynamoDB_20111205.UpdateItem
content-type: application/x-amz-json-1.0

{"TableName":"Table1",
    "Key":
        {"HashKeyElement":{"S":"AttributeValue1"},
        "RangeKeyElement":{"N":"AttributeValue2"}},
    "AttributeUpdates":{"AttributeName3":{"Value":
{"S":"AttributeValue3_New"},"Action":"PUT"}},
    "Expected":{"AttributeName3":{"Value":{"S":"AttributeValue3_Current"}}},
    "ReturnValues":"ReturnValuesConstant"
}
```

Name	Description	Required
`TableName`	The name of the table containing the item to update. Type: String	Yes
`Key`	The primary key that defines the item. For more informati on about primary keys, see Primary key. Type: Map of `HashKeyEl ement` to its value and `RangeKeyElement` to its value.	Yes
`AttributeUpdates`	Map of attribute name to the new value and action for the update. The attribute names specify the attributes to modify, and cannot contain any primary key attributes. Type: Map of attribute name, value, and an action for the attribute update.	
`Attribute Updates :Action`	Specifies how to perform the update. Possible values: `PUT` (default), `ADD` or `DELETE`. The semantics are explained in the UpdateItem descripti on. Type: String	No

Name	Description	Required
	Default: PUT	
`Expected`	Designates an attribute for a conditional update. The `Expected` parameter allows you to provide an attribute name, and whether or not DynamoDB should check to see if the attribute value already exists; or if the attribute value exists and has a particular value before changing it. Type: Map of attribute names.	No
`Expected:AttributeName`	The name of the attribute for the conditional put. Type: String	No

Name	Description	Required

`Expected:AttributeName: ExpectedAttributeValue`	Use this parameter to specify whether or not a value already exists for the attribute name-value pair.
	
	The following JSON notation updates the item if the "Color" attribute doesn't already exist for that item:
	
	`"Expected" : {"Color":{"Exists":false}}`
	
	The following JSON notation checks to see if the attribute with name "Color" has an existing value of "Yellow" before updating the item:
	
	`"Expected" : {"Color":{"Exists":true},{"Value":{"S":"Yellow"}}}`
	
	By default, if you use the `Expected` parameter and provide a `Value`, DynamoDB assumes the attribute exists and has a current value to be replaced. So you don't have to specify `{"Exists":true}`, because it is implied. You can shorten the request to:
	
	`"Expected" :`
		No

Name	Description	Required
	`{"Color":{"Value":{"S":"Yellow"}}}` **Note** If you specify `{"Exists":true}` without an attribute value to check, DynamoDB returns an error.	

Name	Description	Required
ReturnValues	Use this parameter if you want to get the attribute name-value pairs before they were updated with the UpdateItem request. Possible parameter values are NONE (default) or ALL_OLD, UPDATED_OLD , ALL_NEW or UPDATED_NEW . If ALL_OLD is specified, and UpdateItem overwrote an attribute name-value pair, the content of the old item is returned. If this parame ter is not provided or is NONE, nothing is returned . If ALL_NEW is specified, then all the attributes of the new version of the item are returned. If UPDATED_NEW is specified, then the new versions of only the updated attributes are returned. Type: String	No

Responses

Syntax

The following syntax example assumes the request specified a ReturnValues parameter of ALL_OLD; otherwise, the response has only the ConsumedCapacityUnits element.

```
HTTP/1.1 200
x-amzn-RequestId: 8966d095-71e9-11e0-a498-71d736f27375
```

```
content-type: application/x-amz-json-1.0
content-length: 140

{"Attributes":{
 "AttributeName1":{"S":"AttributeValue1"},
 "AttributeName2":{"S":"AttributeValue2"},
 "AttributeName3":{"S":"AttributeValue3"},
 "AttributeName5":{"B":"dmFsdWU="}
 },
"ConsumedCapacityUnits":1
}
```

Name	Description
`Attributes`	A map of attribute name-value pairs, but only if the `ReturnValues` parameter is specified as something other than `NONE` in the request. Type: Map of attribute name-value pairs.
`ConsumedCapacityUnits`	The number of write capacity units consumed by the operation. This value shows the number applied toward your provision ed throughput. For more information see DynamoDB provisioned capacity mode. Type: Number

Special errors

Error	Description
`ConditionalCheckFailedException`	Conditional check failed. Attribute ("+ name +") value is ("+ value +") but was expected ("+ expValue +")
`ResourceNotFoundExceptions`	The specified item or attribute was not found.

Examples

For examples using the AWS SDK, see Working with items and attributes in DynamoDB.

Sample request

```
// This header is abbreviated. For a sample of a complete header, see DynamoDB low-level API.
POST / HTTP/1.1
x-amz-target: DynamoDB_20111205.UpdateItem
content-type: application/x-amz-json-1.0

{"TableName":"comp5",
    "Key":
        {"HashKeyElement":{"S":"Julie"},"RangeKeyElement":{"N":"1307654350"}},
    "AttributeUpdates":
        {"status":{"Value":{"S":"online"},
        "Action":"PUT"}},
    "Expected":{"status":{"Value":{"S":"offline"}}},
    "ReturnValues":"ALL_NEW"
}
```

Sample response

```
HTTP/1.1 200 OK
x-amzn-RequestId: 5IMHO7F01Q9P7Q6QMKMMI3R3QRVV4KQNSO5AEMVJF66Q9ASUAAJG
content-type: application/x-amz-json-1.0
content-length: 121
Date: Fri, 26 Aug 2011 21:05:00 GMT

{"Attributes":
    {"friends":{"SS":["Lynda, Aaron"]},
    "status":{"S":"online"},
    "time":{"N":"1307654350"},
    "user":{"S":"Julie"}},
"ConsumedCapacityUnits":1
}
```

Related actions

- PutItem
- DeleteItem

UpdateTable

> **Important**
>
> ***This section refers to API version 2011-12-05, which is deprecated and should not be used for new applications.***
>
> **For documentation on the current low-level API, see the Amazon DynamoDB API Reference.**

Description

Updates the provisioned throughput for the given table. Setting the throughput for a table helps you manage performance and is part of the provisioned throughput feature of DynamoDB. For more information, see DynamoDB provisioned capacity mode.

The provisioned throughput values can be upgraded or downgraded based on the maximums and minimums listed in Quotas in Amazon DynamoDB.

The table must be in the `ACTIVE` state for this operation to succeed. UpdateTable is an asynchronous operation; while executing the operation, the table is in the `UPDATING` state. While the table is in the `UPDATING` state, the table still has the provisioned throughput from before the call. The new provisioned throughput setting is in effect only when the table returns to the `ACTIVE` state after the UpdateTable operation.

Requests

Syntax

```
// This header is abbreviated.
// For a sample of a complete header, see DynamoDB low-level API.
POST / HTTP/1.1
x-amz-target: DynamoDB_20111205.UpdateTable
content-type: application/x-amz-json-1.0

{"TableName":"Table1",
    "ProvisionedThroughput":{"ReadCapacityUnits":5,"WriteCapacityUnits":15}
}
```

Name	Description	Required
`TableName`	The name of the table to update. Type: String	Yes
`ProvisionedThroughput`	New throughput for the specified table, consisting of values for `ReadCapacityUnits` and `WriteCapacityUnits`. See DynamoDB provisioned capacity mode. Type: Array	Yes
`ProvisionedThroughput:ReadCapacityUnits`	Sets the minimum number of consistent `ReadCapacityUnits` consumed per second for the specified table before DynamoDB balances the load with other operations. Eventually consistent read operations require less effort than a consistent read operation, so a setting of 50 consistent `ReadCapacityUnits` per second provides 100 eventually consistent `ReadCapacityUnits` per second. Type: Number	Yes

Name	Description	Required
`ProvisionedThroughput :WriteCapacityUnits`	Sets the minimum number of `WriteCapacityUnits` consumed per second for the specified table before DynamoDB balances the load with other operations. Type: Number	Yes

Responses

Syntax

```
HTTP/1.1 200 OK
x-amzn-RequestId: CSOC7TJPLR0OOKIRLGOHVAICUFVV4KQNSO5AEMVJF66Q9ASUAAJG
Content-Type: application/json
Content-Length: 311
Date: Tue, 12 Jul 2011 21:31:03 GMT

{"TableDescription":
    {"CreationDateTime":1.321657838135E9,
    "KeySchema":
        {"HashKeyElement":{"AttributeName":"AttributeValue1","AttributeType":"S"},
        "RangeKeyElement":{"AttributeName":"AttributeValue2","AttributeType":"N"}},
    "ProvisionedThroughput":
        {"LastDecreaseDateTime":1.321661704489E9,
        "LastIncreaseDateTime":1.321663607695E9,
        "ReadCapacityUnits":5,
        "WriteCapacityUnits":10},
    "TableName":"Table1",
    "TableStatus":"UPDATING"}}
```

Name	Description
`CreationDateTime`	Date when the table was created.

Name	Description
	Type: Number
KeySchema	The primary key (simple or composite) structure for the table. A name-value pair for the HashKeyElement is required, and a name-value pair for the RangeKeyElement is optional (only required for composite primary keys). The maximum hash key size is 2048 bytes. The maximum range key size is 1024 bytes. Both limits are enforced separatel y (i.e. you can have a combined hash + range 2048 + 1024 key). For more information about primary keys, see Primary key. Type: Map of HashKeyElement , or HashKeyElement and RangeKeyElement for a composite primary key.
ProvisionedThroughput	Current throughput settings for the specified table, including values for LastIncre aseDateTime (if applicable), LastDecre aseDateTime (if applicable), Type: Array
TableName	The name of the updated table. Type: String
TableStatus	The current state of the table (CREATING, ACTIVE, DELETING or UPDATING), which should be UPDATING. Use the DescribeTables operationto check the status of the table. Type: String

Special errors

Error	Description
ResourceNotFoundException	The specified table was not found.
ResourceInUseException	The table is not in the ACTIVE state.

Examples

Sample request

```
// This header is abbreviated.
// For a sample of a complete header, see DynamoDB low-level API.
POST / HTTP/1.1
x-amz-target: DynamoDB_20111205.UpdateTable
content-type: application/x-amz-json-1.0

{"TableName":"comp1",
    "ProvisionedThroughput":{"ReadCapacityUnits":5,"WriteCapacityUnits":15}
}
```

Sample response

```
HTTP/1.1 200 OK
content-type: application/x-amz-json-1.0
content-length: 390
Date: Sat, 19 Nov 2011 00:46:47 GMT

{"TableDescription":
    {"CreationDateTime":1.321657838135E9,
    "KeySchema":
        {"HashKeyElement":{"AttributeName":"user","AttributeType":"S"},
        "RangeKeyElement":{"AttributeName":"time","AttributeType":"N"}},
    "ProvisionedThroughput":
        {"LastDecreaseDateTime":1.321661704489E9,
        "LastIncreaseDateTime":1.321663607695E9,
        "ReadCapacityUnits":5,
        "WriteCapacityUnits":10},
    "TableName":"comp1",
```

```
    "TableStatus":"UPDATING"}
}
```

Related actions

- CreateTable
- DescribeTables
- DeleteTable

Legacy DynamoDB conditional parameters

This document provides an overview of legacy conditional parameters in DynamoDB and recommends using the new expression parameters instead. It covers details on parameters like AttributesToGet, AttributeUpdates, ConditionalOperator, Expected, KeyConditions, QueryFilter, and ScanFilter, and provides examples of how to use the new expression parameters as replacements.

> **Important**
>
> We recommend that you use the new expression parameters instead of these legacy parameters whenever possible. For more information, see Using expressions in DynamoDB. Additionally, DynamoDB does not allow mixing legacy conditional parameters and expression parameters in a single call. For example, calling the `Query` operation with `AttributesToGet` and `ConditionExpression` will result in an error.

The following table shows the DynamoDB API operations that still support these legacy parameters, and which expression parameter to use instead. This table can be helpful if you are considering updating your applications so that they use expression parameters instead.

If you use this API operation ...	With these legacy parameters...	Use this expression parameter instead
`BatchGetItem`	`AttributesToGet`	`ProjectionExpression`
`DeleteItem`	`Expected`	`ConditionExpression`
`GetItem`	`AttributesToGet`	`ProjectionExpression`

If you use this API operation ...	With these legacy parameters...	Use this expression parameter instead
PutItem	Expected	ConditionExpression
Query	AttributesToGet	ProjectionExpression
	KeyConditions	KeyConditionExpression
	QueryFilter	FilterExpression
Scan	AttributesToGet	ProjectionExpression
	ScanFilter	FilterExpression
UpdateItem	AttributeUpdates	UpdateExpression
	Expected	ConditionExpression

The following sections provide more information about legacy conditional parameters.

Topics

- AttributesToGet (legacy)
- AttributeUpdates (legacy)
- ConditionalOperator (legacy)
- Expected (legacy)
- KeyConditions (legacy)
- QueryFilter (legacy)
- ScanFilter (legacy)
- Writing conditions with legacy parameters

AttributesToGet (legacy)

> **Note**
>
> We recommend that you use the new expression parameters instead of these legacy parameters whenever possible. For more information, see Using expressions in DynamoDB. For specific information on the new parameter replacing this one, Use *ProjectionExpression* instead.

The legacy conditional parameter `AttributesToGet` is an array of one or more attributes to retrieve from DynamoDB. If no attribute names are provided, then all attributes will be returned. If any of the requested attributes are not found, they will not appear in the result.

`AttributesToGet` allows you to retrieve attributes of type List or Map; however, it cannot retrieve individual elements within a List or a Map.

Note that `AttributesToGet` has no effect on provisioned throughput consumption. DynamoDB determines capacity units consumed based on item size, not on the amount of data that is returned to an application.

Use *ProjectionExpression* instead – Example

Suppose you wanted to retrieve an item from the *Music* table, but that you only wanted to return some of the attributes. You could use a `GetItem` request with an `AttributesToGet` parameter, as in this AWS CLI example:

```
aws dynamodb get-item \
    --table-name Music \
    --attributes-to-get '["Artist", "Genre"]' \
    --key '{
        "Artist": {"S":"No One You Know"},
        "SongTitle": {"S":"Call Me Today"}
    }'
```

You can use a `ProjectionExpression` instead:

```
aws dynamodb get-item \
    --table-name Music \
    --projection-expression "Artist, Genre" \
```

```
    --key '{
        "Artist": {"S":"No One You Know"},
        "SongTitle": {"S":"Call Me Today"}
    }'
```

AttributeUpdates (legacy)

> **Note**
>
> We recommend that you use the new expression parameters instead of these legacy parameters whenever possible. For more information, see Using expressions in DynamoDB. For specific information on the new parameter replacing this one, use *UpdateExpression* instead..

In an `UpdateItem` operation, the legacy conditional parameter `AttributeUpdates` contains the names of attributes to be modified, the action to perform on each, and the new value for each. If you are updating an attribute that is an index key attribute for any indexes on that table, the attribute type must match the index key type defined in the `AttributesDefinition` of the table description. You can use `UpdateItem` to update any non-key attributes.

Attribute values cannot be null. String and Binary type attributes must have lengths greater than zero. Set type attributes must not be empty. Requests with empty values will be rejected with a `ValidationException` exception.

Each `AttributeUpdates` element consists of an attribute name to modify, along with the following:

- `Value` - The new value, if applicable, for this attribute.
- `Action` - A value that specifies how to perform the update. This action is only valid for an existing attribute whose data type is Number or is a set; do not use ADD for other data types.

  If an item with the specified primary key is found in the table, the following values perform the following actions:

  - PUT - Adds the specified attribute to the item. If the attribute already exists, it is replaced by the new value.
  - `DELETE` - Removes the attribute and its value, if no value is specified for `DELETE`. The data type of the specified value must match the existing value's data type.

If a set of values is specified, then those values are subtracted from the old set. For example, if the attribute value was the set `[a,b,c]` and the `DELETE` action specifies `[a,c]`, then the final attribute value is `[b]`. Specifying an empty set is an error.

- `ADD` - Adds the specified value to the item, if the attribute does not already exist. If the attribute does exist, then the behavior of `ADD` depends on the data type of the attribute:
  - If the existing attribute is a number, and if `Value` is also a number, then `Value` is mathematically added to the existing attribute. If `Value` is a negative number, then it is subtracted from the existing attribute.

    > **Note**
    >
    > If you use `ADD` to increment or decrement a number value for an item that doesn't exist before the update, DynamoDB uses 0 as the initial value.
    > Similarly, if you use `ADD` for an existing item to increment or decrement an attribute value that doesn't exist before the update, DynamoDB uses `0` as the initial value. For example, suppose that the item you want to update doesn't have an attribute named *itemcount*, but you decide to `ADD` the number `3` to this attribute anyway. DynamoDB will create the *itemcount* attribute, set its initial value to `0`, and finally add `3` to it. The result will be a new *itemcount* attribute, with a value of `3`.

  - If the existing data type is a set, and if `Value` is also a set, then `Value` is appended to the existing set. For example, if the attribute value is the set `[1,2]`, and the `ADD` action specified `[3]`, then the final attribute value is `[1,2,3]`. An error occurs if an `ADD` action is specified for a set attribute and the attribute type specified does not match the existing set type.

    Both sets must have the same primitive data type. For example, if the existing data type is a set of strings, `Value` must also be a set of strings.

If no item with the specified key is found in the table, the following values perform the following actions:

- `PUT` - Causes DynamoDB to create a new item with the specified primary key, and then adds the attribute.
- `DELETE` - Nothing happens, because attributes cannot be deleted from a nonexistent item. The operation succeeds, but DynamoDB does not create a new item.
- `ADD` - Causes DynamoDB to create an item with the supplied primary key and number (or set of numbers) for the attribute value. The only data types allowed are Number and Number Set.

If you provide any attributes that are part of an index key, then the data types for those attributes must match those of the schema in the table's attribute definition.

Use *UpdateExpression* instead – Example

Suppose you wanted to modify an item in the *Music* table. You could use an `UpdateItem` request with an `AttributeUpdates` parameter, as in this AWS CLI example:

```
aws dynamodb update-item \
    --table-name Music \
    --key '{
        "SongTitle": {"S":"Call Me Today"},
        "Artist": {"S":"No One You Know"}
    }' \
    --attribute-updates '{
        "Genre": {
            "Action": "PUT",
            "Value": {"S":"Rock"}
        }
    }'
```

You can use a `UpdateExpression` instead:

```
aws dynamodb update-item \
    --table-name Music \
    --key '{
        "SongTitle": {"S":"Call Me Today"},
        "Artist": {"S":"No One You Know"}
    }' \
    --update-expression 'SET Genre = :g' \
    --expression-attribute-values '{
        ":g": {"S":"Rock"}
    }'
```

ConditionalOperator (legacy)

> **Note**
>
> We recommend that you use the new expression parameters instead of these legacy parameters whenever possible. For more information, see Using expressions in DynamoDB.

The legacy conditional parameter `ConditionalOperator` is a logical operator used to apply to the conditions in a `Expected`, `QueryFilter` or `ScanFilter` map:

- AND - If all of the conditions evaluate to true, then the entire map evaluates to true.
- OR - If at least one of the conditions evaluates to true, then the entire map evaluates to true.

If you omit `ConditionalOperator`, then `AND` is the default.

The operation will succeed only if the entire map evaluates to true.

> **Note**
>
> This parameter does not support attributes of type List or Map.

Expected (legacy)

> **Note**
>
> We recommend that you use the new expression parameters instead of these legacy parameters whenever possible. For more information, see Using expressions in DynamoDB. For specific information on the new parameter replacing this one, use *ConditionExpression* instead..

The legacy conditional parameter `Expected` is a conditional block for an `UpdateItem` operation. `Expected` is a map of attribute/condition pairs. Each element of the map consists of an attribute name, a comparison operator, and one or more values. DynamoDB compares the attribute with the value(s) you supplied, using the comparison operator. For each `Expected` element, the result of the evaluation is either true or false.

If you specify more than one element in the `Expected` map, then by default all of the conditions must evaluate to true. In other words, the conditions are combined using `AND` operator. (You can use the `ConditionalOperator` parameter to OR the conditions instead. If you do this, then at least one of the conditions must evaluate to true, rather than all of them.)

If the `Expected` map evaluates to true, then the conditional operation succeeds; otherwise, it fails.

`Expected` contains the following:

- `AttributeValueList` - One or more values to evaluate against the supplied attribute. The number of values in the list depends on the `ComparisonOperator` being used.

  For type Number, value comparisons are numeric.

  String value comparisons for greater than, equals, or less than are based on Unicode with UTF-8 binary encoding. For example, `a` is greater than `A`, and `a` is greater than `B`.

  For type Binary, DynamoDB treats each byte of the binary data as unsigned when it compares binary values.

- `ComparisonOperator` - A comparator for evaluating attributes in the `AttributeValueList`. When performing the comparison, DynamoDB uses strongly consistent reads.

  The following comparison operators are available:

  `EQ | NE | LE | LT | GE | GT | NOT_NULL | NULL | CONTAINS | NOT_CONTAINS | BEGINS_WITH | IN | BETWEEN`

  The following are descriptions of each comparison operator.

  - `EQ` : Equal. `EQ` is supported for all data types, including lists and maps.

    `AttributeValueList` can contain only one `AttributeValue` element of type String, Number, Binary, String Set, Number Set, or Binary Set. If an item contains an `AttributeValue` element of a different type than the one provided in the request, the value does not match. For example, `{"S":"6"}` does not equal `{"N":"6"}`. Also, `{"N":"6"}` does not equal `{"NS":["6", "2", "1"]}`.

  - `NE` : Not equal. `NE` is supported for all data types, including lists and maps.

    `AttributeValueList` can contain only one `AttributeValue` of type String, Number, Binary, String Set, Number Set, or Binary Set. If an item contains an `AttributeValue` of a different type than the one provided in the request, the value does not match. For example, `{"S":"6"}` does not equal `{"N":"6"}`. Also, `{"N":"6"}` does not equal `{"NS":["6", "2", "1"]}`.

  - `LE` : Less than or equal.

    `AttributeValueList` can contain only one `AttributeValue` element of type String, Number, or Binary (not a set type). If an item contains an `AttributeValue` element of a

different type than the one provided in the request, the value does not match. For example, `{"S":"6"}` does not equal `{"N":"6"}`. Also, `{"N":"6"}` does not compare to `{"NS":["6", "2", "1"]}`.

- `LT` : Less than.

  `AttributeValueList` can contain only one `AttributeValue` of type String, Number, or Binary (not a set type). If an item contains an `AttributeValue` element of a different type than the one provided in the request, the value does not match. For example, `{"S":"6"}` does not equal `{"N":"6"}`. Also, `{"N":"6"}` does not compare to `{"NS":["6", "2", "1"]}`.

- `GE` : Greater than or equal.

  `AttributeValueList` can contain only one `AttributeValue` element of type String, Number, or Binary (not a set type). If an item contains an `AttributeValue` element of a different type than the one provided in the request, the value does not match. For example, `{"S":"6"}` does not equal `{"N":"6"}`. Also, `{"N":"6"}` does not compare to `{"NS":["6", "2", "1"]}`.

- `GT` : Greater than.

  `AttributeValueList` can contain only one `AttributeValue` element of type String, Number, or Binary (not a set type). If an item contains an `AttributeValue` element of a different type than the one provided in the request, the value does not match. For example, `{"S":"6"}` does not equal `{"N":"6"}`. Also, `{"N":"6"}` does not compare to `{"NS":["6", "2", "1"]}`.

- `NOT_NULL` : The attribute exists. `NOT_NULL` is supported for all data types, including lists and maps.

  > **Note**
  >
  > This operator tests for the existence of an attribute, not its data type. If the data type of attribute "a" is null, and you evaluate it using `NOT_NULL`, the result is a Boolean `true`. This result is because the attribute "a" exists; its data type is not relevant to the `NOT_NULL` comparison operator.

- `NULL` : The attribute does not exist. `NULL` is supported for all data types, including lists and maps.

> **Note**
>
> This operator tests for the nonexistence of an attribute, not its data type. If the data type of attribute "a" is null, and you evaluate it using NULL, the result is a Boolean `false`. This is because the attribute "a" exists; its data type is not relevant to the NULL comparison operator.

- `CONTAINS` : Checks for a subsequence, or value in a set.

  `AttributeValueList` can contain only one `AttributeValue` element of type String, Number, or Binary (not a set type). If the target attribute of the comparison is of type String, then the operator checks for a substring match. If the target attribute of the comparison is of type Binary, then the operator looks for a subsequence of the target that matches the input. If the target attribute of the comparison is a set ("SS", "NS", or "BS"), then the operator evaluates to true if it finds an exact match with any member of the set.

  CONTAINS is supported for lists: When evaluating "a `CONTAINS` b", " a" can be a list; however, "b" cannot be a set, a map, or a list.

- `NOT_CONTAINS` : Checks for absence of a subsequence, or absence of a value in a set.

  `AttributeValueList` can contain only one `AttributeValue` element of type String, Number, or Binary (not a set type). If the target attribute of the comparison is a String, then the operator checks for the absence of a substring match. If the target attribute of the comparison is Binary, then the operator checks for the absence of a subsequence of the target that matches the input. If the target attribute of the comparison is a set ("SS", "NS", or "BS"), then the operator evaluates to true if it `does not` find an exact match with any member of the set.

  NOT_CONTAINS is supported for lists: When evaluating "a `NOT CONTAINS` b", " a" can be a list; however, "b" cannot be a set, a map, or a list.

- `BEGINS_WITH` : Checks for a prefix.

  `AttributeValueList` can contain only one `AttributeValue` of type String or Binary (not a Number or a set type). The target attribute of the comparison must be of type String or Binary (not a Number or a set type).

- `IN` : Checks for matching elements within two sets.

`AttributeValueList` can contain one or more `AttributeValue` elements of type String, Number, or Binary (not a set type). These attributes are compared against an existing set type attribute of an item. If any elements of the input set are present in the item attribute, the expression evaluates to true.

- `BETWEEN` : Greater than or equal to the first value, and less than or equal to the second value.

  `AttributeValueList` must contain two `AttributeValue` elements of the same type, either String, Number, or Binary (not a set type). A target attribute matches if the target value is greater than, or equal to, the first element and less than, or equal to, the second element. If an item contains an `AttributeValue` element of a different type than the one provided in the request, the value does not match. For example, `{"S":"6"}` does not compare to `{"N":"6"}`. Also, `{"N":"6"}` does not compare to `{"NS":["6", "2", "1"]}`

The following parameters can be used instead of `AttributeValueList` and `ComparisonOperator`:

- `Value` - A value for DynamoDB to compare with an attribute.
- `Exists` - A Boolean value that causes DynamoDB to evaluate the value before attempting the conditional operation:
  - If `Exists` is `true`, DynamoDB will check to see if that attribute value already exists in the table. If it is found, then the condition evaluates to true; otherwise the condition evaluates to false.
  - If `Exists` is `false`, DynamoDB assumes that the attribute value does not exist in the table. If in fact the value does not exist, then the assumption is valid and the condition evaluates to true. If the value is found, despite the assumption that it does not exist, the condition evaluates to false.

  Note that the default value for `Exists` is `true`.

The `Value` and `Exists` parameters are incompatible with `AttributeValueList` and `ComparisonOperator`. Note that if you use both sets of parameters at once, DynamoDB will return a `ValidationException` exception.

> **Note**
>
> This parameter does not support attributes of type List or Map.

Use *ConditionExpression* instead – Example

Suppose you wanted to modify an item in the *Music* table, but only if a certain condition was true. You could use an `UpdateItem` request with an `Expected` parameter, as in this AWS CLI example:

```
aws dynamodb update-item \
    --table-name Music \
    --key '{
        "Artist": {"S":"No One You Know"},
        "SongTitle": {"S":"Call Me Today"}
    }' \
    --attribute-updates '{
        "Price": {
            "Action": "PUT",
            "Value": {"N":"1.98"}
        }
    }' \
    --expected '{
        "Price": {
            "ComparisonOperator": "LE",
            "AttributeValueList": [ {"N":"2.00"} ]
        }
    }'
```

You can use a `ConditionExpression` instead:

```
aws dynamodb update-item \
    --table-name Music \
    --key '{
        "Artist": {"S":"No One You Know"},
        "SongTitle": {"S":"Call Me Today"}
    }' \
    --update-expression 'SET Price = :p1' \
    --condition-expression 'Price <= :p2' \
    --expression-attribute-values '{
        ":p1": {"N":"1.98"},
        ":p2": {"N":"2.00"}
```

```
    }'
```

KeyConditions (legacy)

> **Note**
>
> We recommend that you use the new expression parameters instead of these legacy parameters whenever possible. For more information, see Using expressions in DynamoDB. For specific information on the new parameter replacing this one, use *KeyConditionExpression* instead..

The legacy conditional parameter `KeyConditions` contains selection criteria for a `Query` operation. For a query on a table, you can have conditions only on the table primary key attributes. You must provide the partition key name and value as an `EQ` condition. You can optionally provide a second condition, referring to the sort key.

> **Note**
>
> If you don't provide a sort key condition, all of the items that match the partition key will be retrieved. If a `FilterExpression` or `QueryFilter` is present, it will be applied after the items are retrieved.

For a query on an index, you can have conditions only on the index key attributes. You must provide the index partition key name and value as an `EQ` condition. You can optionally provide a second condition, referring to the index sort key.

Each `KeyConditions` element consists of an attribute name to compare, along with the following:

- `AttributeValueList` - One or more values to evaluate against the supplied attribute. The number of values in the list depends on the `ComparisonOperator` being used.

  For type Number, value comparisons are numeric.

  String value comparisons for greater than, equals, or less than are based on Unicode with UTF-8 binary encoding. For example, a is greater than A, and a is greater than B.

For Binary, DynamoDB treats each byte of the binary data as unsigned when it compares binary values.

- `ComparisonOperator` - A comparator for evaluating attributes. For example: equals, greater than, and less than.

  For `KeyConditions`, only the following comparison operators are supported:

  `EQ | LE | LT | GE | GT | BEGINS_WITH | BETWEEN`

  The following are descriptions of these comparison operators.

  - `EQ` : Equal.

    `AttributeValueList` can contain only one `AttributeValue` of type String, Number, or Binary (not a set type). If an item contains an `AttributeValue` element of a different type than the one specified in the request, the value does not match. For example, `{"S":"6"}` does not equal `{"N":"6"}`. Also, `{"N":"6"}` does not equal `{"NS":["6", "2", "1"]}`.

  - `LE` : Less than or equal.

    `AttributeValueList` can contain only one `AttributeValue` element of type String, Number, or Binary (not a set type). If an item contains an `AttributeValue` element of a different type than the one provided in the request, the value does not match. For example, `{"S":"6"}` does not equal `{"N":"6"}`. Also, `{"N":"6"}` does not compare to `{"NS":["6", "2", "1"]}`.

  - `LT` : Less than.

    `AttributeValueList` can contain only one `AttributeValue` of type String, Number, or Binary (not a set type). If an item contains an `AttributeValue` element of a different type than the one provided in the request, the value does not match. For example, `{"S":"6"}` does not equal `{"N":"6"}`. Also, `{"N":"6"}` does not compare to `{"NS":["6", "2", "1"]}`.

  - `GE` : Greater than or equal.

    `AttributeValueList` can contain only one `AttributeValue` element of type String, Number, or Binary (not a set type). If an item contains an `AttributeValue` element of a different type than the one provided in the request, the value does not match. For example, `{"S":"6"}` does not equal `{"N":"6"}`. Also, `{"N":"6"}` does not compare to `{"NS":["6", "2", "1"]}`.

  - `GT` : Greater than.

`AttributeValueList` can contain only one `AttributeValue` element of type String, Number, or Binary (not a set type). If an item contains an `AttributeValue` element of a different type than the one provided in the request, the value does not match. For example, `{"S":"6"}` does not equal `{"N":"6"}`. Also, `{"N":"6"}` does not compare to `{"NS":["6", "2", "1"]}`.

- `BEGINS_WITH` : Checks for a prefix.

  `AttributeValueList` can contain only one `AttributeValue` of type String or Binary (not a Number or a set type). The target attribute of the comparison must be of type String or Binary (not a Number or a set type).

- `BETWEEN` : Greater than or equal to the first value, and less than or equal to the second value.

  `AttributeValueList` must contain two `AttributeValue` elements of the same type, either String, Number, or Binary (not a set type). A target attribute matches if the target value is greater than, or equal to, the first element and less than, or equal to, the second element. If an item contains an `AttributeValue` element of a different type than the one provided in the request, the value does not match. For example, `{"S":"6"}` does not compare to `{"N":"6"}`. Also, `{"N":"6"}` does not compare to `{"NS":["6", "2", "1"]}`.

Use *KeyConditionExpression* instead – Example

Suppose you wanted to retrieve several items with the same partition key from the *Music* table. You could use a `Query` request with a `KeyConditions` parameter, as in this AWS CLI example:

```
aws dynamodb query \
    --table-name Music \
    --key-conditions '{
        "Artist":{
            "ComparisonOperator":"EQ",
            "AttributeValueList": [ {"S": "No One You Know"} ]
        },
        "SongTitle":{
            "ComparisonOperator":"BETWEEN",
            "AttributeValueList": [ {"S": "A"}, {"S": "M"} ]
        }
    }'
```

You can use a `KeyConditionExpression` instead:

```
aws dynamodb query \
    --table-name Music \
    --key-condition-expression 'Artist = :a AND SongTitle BETWEEN :t1 AND :t2' \
    --expression-attribute-values '{
        ":a": {"S": "No One You Know"},
        ":t1": {"S": "A"},
        ":t2": {"S": "M"}
    }'
```

QueryFilter (legacy)

> **Note**
>
> We recommend that you use the new expression parameters instead of these legacy parameters whenever possible. For more information, see Using expressions in DynamoDB. For specific information on the new parameter replacing this one, use *FilterExpression* instead..

In a `Query` operation, the legacy conditional parameter `QueryFilter` is a condition that evaluates the query results after the items are read and returns only the desired values.

This parameter does not support attributes of type List or Map.

> **Note**
>
> A `QueryFilter` is applied after the items have already been read; the process of filtering does not consume any additional read capacity units.

If you provide more than one condition in the `QueryFilter` map, then by default all of the conditions must evaluate to true. In other words, the conditions are combined using the `AND` operator. (You can use the ConditionalOperator (legacy) parameter to OR the conditions instead. If you do this, then at least one of the conditions must evaluate to true, rather than all of them.)

Note that `QueryFilter` does not allow key attributes. You cannot define a filter condition on a partition key or a sort key.

Each `QueryFilter` element consists of an attribute name to compare, along with the following:

- `AttributeValueList` - One or more values to evaluate against the supplied attribute. The number of values in the list depends on the operator specified in `ComparisonOperator`.

  For type Number, value comparisons are numeric.

  String value comparisons for greater than, equals, or less than are based on UTF-8 binary encoding. For example, a is greater than A, and a is greater than B.

  For type Binary, DynamoDB treats each byte of the binary data as unsigned when it compares binary values.

  For information on specifying data types in JSON, see DynamoDB low-level API.
- `ComparisonOperator` - A comparator for evaluating attributes. For example: equals, greater than, and less than.

  The following comparison operators are available:

  `EQ | NE | LE | LT | GE | GT | NOT_NULL | NULL | CONTAINS | NOT_CONTAINS | BEGINS_WITH | IN | BETWEEN`

Use *FilterExpression* instead – Example

Suppose you wanted to query the *Music* table and apply a condition to the matching items. You could use a `Query` request with a `QueryFilter` parameter, as in this AWS CLI example:

```
aws dynamodb query \
    --table-name Music \
    --key-conditions '{
        "Artist": {
            "ComparisonOperator": "EQ",
            "AttributeValueList": [ {"S": "No One You Know"} ]
        }
    }' \
    --query-filter '{
        "Price": {
            "ComparisonOperator": "GT",
            "AttributeValueList": [ {"N": "1.00"} ]
        }
    }'
```

You can use a `FilterExpression` instead:

```
aws dynamodb query \
    --table-name Music \
    --key-condition-expression 'Artist = :a' \
    --filter-expression 'Price > :p' \
    --expression-attribute-values '{
        ":p": {"N":"1.00"},
        ":a": {"S":"No One You Know"}
    }'
```

ScanFilter (legacy)

> **Note**
>
> We recommend that you use the new expression parameters instead of these legacy parameters whenever possible. For more information, see Using expressions in DynamoDB. For specific information on the new parameter replacing this one, use *FilterExpression* instead..

In a `Scan` operation, the legacy conditional parameter `ScanFilter` is a condition that evaluates the scan results and returns only the desired values.

> **Note**
>
> This parameter does not support attributes of type List or Map.

If you specify more than one condition in the `ScanFilter` map, then by default all of the conditions must evaluate to true. In other words, the conditions are ANDed together. (You can use the ConditionalOperator (legacy) parameter to OR the conditions instead. If you do this, then at least one of the conditions must evaluate to true, rather than all of them.)

Each `ScanFilter` element consists of an attribute name to compare, along with the following:

- `AttributeValueList` - One or more values to evaluate against the supplied attribute. The number of values in the list depends on the operator specified in `ComparisonOperator` .

  For type Number, value comparisons are numeric.

String value comparisons for greater than, equals, or less than are based on UTF-8 binary encoding. For example, a is greater than A, and a is greater than B.

For Binary, DynamoDB treats each byte of the binary data as unsigned when it compares binary values.

For information on specifying data types in JSON, see DynamoDB low-level API.

- `ComparisonOperator` - A comparator for evaluating attributes. For example: equals, greater than, and less than.

  The following comparison operators are available:

  `EQ | NE | LE | LT | GE | GT | NOT_NULL | NULL | CONTAINS | NOT_CONTAINS | BEGINS_WITH | IN | BETWEEN`

Use *FilterExpression* instead – Example

Suppose you wanted to scan the *Music* table and apply a condition to the matching items. You could use a `Scan` request with a `ScanFilter` parameter, as in this AWS CLI example:

```
aws dynamodb scan \
    --table-name Music \
    --scan-filter '{
        "Genre":{
            "AttributeValueList":[ {"S":"Rock"} ],
            "ComparisonOperator": "EQ"
        }
    }'
```

You can use a `FilterExpression` instead:

```
aws dynamodb scan \
    --table-name Music \
    --filter-expression 'Genre = :g' \
    --expression-attribute-values '{
        ":g": {"S":"Rock"}
    }'
```

Writing conditions with legacy parameters

> **Note**
>
> We recommend that you use the new expression parameters instead of these legacy parameters whenever possible. For more information, see Using expressions in DynamoDB.

The following section describes how to write conditions for use with legacy parameters, such as `Expected`, `QueryFilter`, and `ScanFilter`.

> **Note**
>
> New applications should use expression parameters instead. For more information, see Using expressions in DynamoDB.

Simple conditions

With attribute values, you can write conditions for comparisons against table attributes. A condition always evaluates to true or false, and consists of:

- `ComparisonOperator` — greater than, less than, equal to, and so on.
- `AttributeValueList` (optional) — attribute value(s) to compare against. Depending on the `ComparisonOperator` being used, the `AttributeValueList` might contain one, two, or more values; or it might not be present at all.

The following sections describe the various comparison operators, along with examples of how to use them in conditions.

Comparison operators with no attribute values

- `NOT_NULL` - true if an attribute exists.
- `NULL` - true if an attribute does not exist.

Use these operators to check whether an attribute exists, or doesn't exist. Because there is no value to compare against, do not specify `AttributeValueList`.

Example

The following expression evaluates to true if the *Dimensions* attribute exists.

```
...
    "Dimensions": {
         ComparisonOperator: "NOT_NULL"
    }
...
```

Comparison operators with one attribute value

- `EQ` - true if an attribute is equal to a value.

  `AttributeValueList` can contain only one value of type String, Number, Binary, String Set, Number Set, or Binary Set. If an item contains a value of a different type than the one specified in the request, the value does not match. For example, the string `"3"` is not equal to the number `3`. Also, the number `3` is not equal to the number set `[3, 2, 1]`.
- `NE` - true if an attribute is not equal to a value.

  `AttributeValueList` can contain only one value of type String, Number, Binary, String Set, Number Set, or Binary Set. If an item contains a value of a different type than the one specified in the request, the value does not match.
- `LE` - true if an attribute is less than or equal to a value.

  `AttributeValueList` can contain only one value of type String, Number, or Binary (not a set). If an item contains an `AttributeValue` of a different type than the one specified in the request, the value does not match.
- `LT` - true if an attribute is less than a value.

  `AttributeValueList` can contain only one value of type String, Number, or Binary (not a set). If an item contains a value of a different type than the one specified in the request, the value does not match.
- `GE` - true if an attribute is greater than or equal to a value.

  `AttributeValueList` can contain only one value of type String, Number, or Binary (not a set). If an item contains a value of a different type than the one specified in the request, the value does not match.
- `GT` - true if an attribute is greater than a value.

`AttributeValueList` can contain only one value of type String, Number, or Binary (not a set). If an item contains a value of a different type than the one specified in the request, the value does not match.

- `CONTAINS` - true if a value is present within a set, or if one value contains another.

  `AttributeValueList` can contain only one value of type String, Number, or Binary (not a set). If the target attribute of the comparison is a String, then the operator checks for a substring match. If the target attribute of the comparison is Binary, then the operator looks for a subsequence of the target that matches the input. If the target attribute of the comparison is a set, then the operator evaluates to true if it finds an exact match with any member of the set.

- `NOT_CONTAINS` - true if a value is *not* present within a set, or if one value does not contain another value.

  `AttributeValueList` can contain only one value of type String, Number, or Binary (not a set). If the target attribute of the comparison is a String, then the operator checks for the absence of a substring match. If the target attribute of the comparison is Binary, then the operator checks for the absence of a subsequence of the target that matches the input. If the target attribute of the comparison is a set, then the operator evaluates to true if it *does not* find an exact match with any member of the set.

- `BEGINS_WITH` - true if the first few characters of an attribute match the provided value. Do not use this operator for comparing numbers.

  `AttributeValueList` can contain only one value of type String or Binary (not a Number or a set). The target attribute of the comparison must be a String or Binary (not a Number or a set).

Use these operators to compare an attribute with a value. You must specify an `AttributeValueList` consisting of a single value. For most of the operators, this value must be a scalar; however, the `EQ` and `NE` operators also support sets.

Examples

The following expressions evaluate to true if:

- A product's price is greater than 100.

```
...
    "Price": {
```

```
        ComparisonOperator: "GT",
        AttributeValueList: [ {"N":"100"} ]
    }
...
```

- A product category begins with "Bo".

```
...
    "ProductCategory": {
        ComparisonOperator: "BEGINS_WITH",
        AttributeValueList: [ {"S":"Bo"} ]
    }
...
```

- A product is available in either red, green, or black:

```
...
    "Color": {
        ComparisonOperator: "EQ",
        AttributeValueList: [
            [ {"S":"Black"}, {"S":"Red"}, {"S":"Green"} ]
        ]
    }
...
```

> **Note**
>
> When comparing set data types, the order of the elements does not matter. DynamoDB will return only the items with the same set of values, regardless of the order in which you specify them in your request.

Comparison operators with two attribute values

- `BETWEEN` - true if a value is between a lower bound and an upper bound, endpoints inclusive.

  `AttributeValueList` must contain two elements of the same type, either String, Number, or Binary (not a set). A target attribute matches if the target value is greater than, or equal to, the first element and less than, or equal to, the second element. If an item contains a value of a different type than the one specified in the request, the value does not match.

Use this operator to determine if an attribute value is within a range. The `AttributeValueList` must contain two scalar elements of the same type - String, Number, or Binary.

Example

The following expression evaluates to true if a product's price is between 100 and 200.

```
...
    "Price": {
        ComparisonOperator: "BETWEEN",
        AttributeValueList: [ {"N":"100"}, {"N":"200"} ]
    }
...
```

Comparison operators with *n* attribute values

- `IN` - true if a value is equal to any of the values in an enumerated list. Only scalar values are supported in the list, not sets. The target attribute must be of the same type and exact value in order to match.

  `AttributeValueList` can contain one or more elements of type String, Number, or Binary (not a set). These attributes are compared against an existing non-set type attribute of an item. If *any* elements of the input set are present in the item attribute, the expression evaluates to true.

  `AttributeValueList` can contain one or more values of type String, Number, or Binary (not a set). The target attribute of the comparison must be of the same type and exact value to match. A String never matches a String set.

Use this operator to determine whether the supplied value is within an enumerated list. You can specify any number of scalar values in `AttributeValueList`, but they all must be of the same data type.

Example

The following expression evaluates to true if the value for *Id* is 201, 203, or 205.

```
...
    "Id": {
        ComparisonOperator: "IN",
        AttributeValueList: [ {"N":"201"}, {"N":"203"}, {"N":"205"} ]
```

```
    }
...
```

Using multiple conditions

DynamoDB lets you combine multiple conditions to form complex expressions. You do this by providing at least two expressions, with an optional ConditionalOperator (legacy).

By default, when you specify more than one condition, *all* of the conditions must evaluate to true in order for the entire expression to evaluate to true. In other words, an implicit *AND* operation takes place.

Example

The following expression evaluates to true if a product is a book that has at least 600 pages. Both of the conditions must evaluate to true, since they are implicitly *AND*ed together.

```
...
    "ProductCategory": {
        ComparisonOperator: "EQ",
        AttributeValueList: [ {"S":"Book"} ]
    },
    "PageCount": {
        ComparisonOperator: "GE",
        AttributeValueList: [ {"N":600} ]
    }
...
```

You can use ConditionalOperator (legacy) to clarify that an *AND* operation will take place. The following example behaves in the same manner as the previous one.

```
...
    "ConditionalOperator" : "AND",
    "ProductCategory": {
        "ComparisonOperator": "EQ",
        "AttributeValueList": [ {"N":"Book"} ]
    },
    "PageCount": {
        "ComparisonOperator": "GE",
        "AttributeValueList": [ {"N":600} ]
    }
```

```
...
```

You can also set `ConditionalOperator` to *OR*, which means that *at least one* of the conditions must evaluate to true.

Example

The following expression evaluates to true if a product is a mountain bike, if it is a particular brand name, or if its price is greater than 100.

```
...
    ConditionalOperator : "OR",
    "BicycleType": {
        "ComparisonOperator": "EQ",
        "AttributeValueList": [ {"S":"Mountain" ]
    },
    "Brand": {
        "ComparisonOperator": "EQ",
        "AttributeValueList": [ {"S":"Brand-Company A" ]
    },
    "Price": {
        "ComparisonOperator": "GT",
        "AttributeValueList": [ {"N":"100"} ]
    }
...
```

Note

In a complex expression, the conditions are processed in order, from the first condition to the last.
You cannot use both AND and OR in a single expression.

Other conditional operators

In previous releases of DynamoDB, the `Expected` parameter behaved differently for conditional writes. Each item in the `Expected` map represented an attribute name for DynamoDB to check, along with the following:

- `Value` — a value to compare against the attribute.
- `Exists` — determine whether the value exists prior to attempting the operation.

The `Value` and `Exists` options continue to be supported in DynamoDB; however, they only let you test for an equality condition, or whether an attribute exists. We recommend that you use `ComparisonOperator` and `AttributeValueList` instead, because these options let you construct a much wider range of conditions.

Example

A `DeleteItem` can check to see whether a book is no longer in publication, and only delete it if this condition is true. Here is an AWS CLI example using a legacy condition:

```
aws dynamodb delete-item \
    --table-name ProductCatalog \
    --key '{
        "Id": {"N":"600"}
    }' \
    --expected '{
        "InPublication": {
            "Exists": true,
            "Value": {"BOOL":false}
        }
    }'
```

The following example does the same thing, but does not use a legacy condition:

```
aws dynamodb delete-item \
    --table-name ProductCatalog \
    --key '{
        "Id": {"N":"600"}
    }' \
    --expected '{
        "InPublication": {
            "ComparisonOperator": "EQ",
            "AttributeValueList": [ {"BOOL":false} ]
        }
    }'
```

Example

A `PutItem` operation can protect against overwriting an existing item with the same primary key attributes. Here is an example using a legacy condition:

```
aws dynamodb put-item \
    --table-name ProductCatalog \
    --item '{
      "Id": {"N":"500"},
        "Title": {"S":"Book 500 Title"}
    }' \
    --expected '{
        "Id": { "Exists": false }
    }'
```

The following example does the same thing, but does not use a legacy condition:

```
aws dynamodb put-item \
    --table-name ProductCatalog \
    --item '{
      "Id": {"N":"500"},
        "Title": {"S":"Book 500 Title"}
    }' \
    --expected '{
        "Id": { "ComparisonOperator": "NULL" }
    }'
```

Note

For conditions in the `Expected` map, do not use the legacy `Value` and `Exists` options together with `ComparisonOperator` and `AttributeValueList`. If you do this, your conditional write will fail.